Missouri University of Science and Technology
Scholars' Mine

Computer Science Technical Reports | Computer Science

01 Dec 1988

A Parallel Implementation of Stickel's AC Unification Algorithm in a Message-Passing Environment

David John Kleikamp

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

Follow this and additional works at: https://scholarsmine.mst.edu/comsci_techreports

Part of the Computer Sciences Commons

Recommended Citation

Kleikamp, David John and Wilkerson, Ralph W., "A Parallel Implementation of Stickel's AC Unification Algorithm in a Message-Passing Environment" (1988). *Computer Science Technical Reports*. 76.
https://scholarsmine.mst.edu/comsci_techreports/76

This Technical Report is brought to you for free and open access by Scholars' Mine. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Scholars' Mine. This work is protected by U. S. Copyright Law. Unauthorized use including reproduction for redistribution requires the permission of the copyright holder. For more information, please contact scholarsmine@mst.edu.

A PARALLEL IMPLEMENTATION OF STICKEL'S AC UNIFICATION ALGORITHM IN A MESSAGE-PASSING ENVIRONMENT

D. J. Kleikamp* and R. W. Wilkerson

CSc-88-11

Department of Computer Science

University of Missouri-Rolla

Rolla, Missouri 65401 (314)341-4491

*This report is substantially the M.S. thesis of the first author, completed, December 1988.

ABSTRACT

Unification algorithms are an essential component of automated reasoning and term rewriting systems. Unification finds a set of substitutions or unifiers that, when applied to variables in two or more terms, make those terms identical or equivalent. Most systems use Robinson's unification algorithm or some variant of it. However, terms containing functions exhibiting properties such as associativity and commutativity may be made equivalent without appearing identical. Systems employing Robinson's unification algorithm must use some mechanism separate from the unification algorithm to reason with such functions. Often this is done by incorporating the properties into a rule base and generating equivalent terms which can be unified by Robinson's algorithm. However, rewriting the terms in this manner can generate large numbers of useless terms in the problem space of the system.

If the properties of the functions are incorporated into the unification algorithm itself, there is no need to rewrite the terms such that they appear identical. Stickel developed an algorithm to unify two terms containing associative and commutative functions. The unifiers (there may be more than one) are found by creating a homogeneous linear Diophantine equation with integer coefficients from the terms being unified. The unifiers can be constructed from solutions to this equation.

The unifiers generated from one solution of the Diophantine equation are independent of any other solution to the equation. Therefore, once the Diophantine equation has been solved, the unifiers can be calculated from the solutions in parallel. We have implemented Stickel's AC unification algorithm to run in parallel on a local area network of Sun 4/110 workstations in an effort to improve the speed of AC unification.

TABLE OF CONTENTS

Page

ABSTRACT ii

ACKNOWLEDGEMENTS iii

LIST OF ILLUSTRATIONS vi

LIST OF TABLES vii

I. INTRODUCTION 1

A. STRUCTURE 1

B. GOALS AND MOTIVATION 2

II. DEFINITIONS 4

A. TERMS 4

B. SUBSTITUTIONS 4

C. PREDICATES AND FIRST-ORDER LOGIC 6

III. UNIFICATION 7

A. APPLICATIONS 7

B. ROBINSON UNIFICATION 9

C. E-UNIFICATION 12

IV. STICKEL'S AC UNIFICATION ALGORITHM 14

A. THE VARIABLE-ONLY CASE 14

B. GENERAL CASE 17

C. SOLVING THE DIOPHANTINE EQUATION 18

1. Huet's Algorithm 18

2. Lankford's Algorithm 19

3. Zhang's Algorithm 20

V. PARALLEL COMPUTING 22

A. PARALLEL HARDWARE 22

B. PARALLEL SOFTWARE 23

C. ARGONNE'S TOOLS FOR PORTABLE PARALLEL PROGRAMS 25

1. The Shared Memory Model 26

2. The Message Passing Model 28

VI. SEQUENTIAL IMPLEMENTATION 30

A. DATA STRUCTURES 30

B. ROBINSON'S UNIFICATION ALGORITHM 31

C. STICKEL'S ALGORITHM 32

D. OTHER CONSIDERATIONS 32

VII. PARALLEL IMPLEMENTATION 34

A. OPPORTUNITIES FOR EXPLOITING PARALLELISM . . . 34

B. IMPLEMENTATION 35

VIII. RESULTS 39

A. OBSERVATIONS 39

B. ANALYSIS OF SEQUENTIAL PROGRAM 40

C. TOPICS FOR FUTURE RESEARCH 41

REFERENCES 44

VITA 46

APPENDIX: Source Listing - Parallel Version 47

LIST OF ILLUSTRATIONS

Figure		Page
1	The Composition of Two Substitutions	5
2	Unifiers and mgu's	7
3	Robinson's Unification Algorithm	10
4	An example of Robinson's unification algorithm	11
5	Stickel's Variable-Only AC Unification Algorithm	16
6	Stickel's General AC Unification Algorithm	18
7	A Monitor For Sending and Receiving Messages	27
8	AC Unification Algorithm - Master	37
9	AC Unification Algorithm - Slave	38

LIST OF TABLES

Table Page

I BASIS OF SOLUTIONS TO DIOPHANTINE EQUATIONS 15

II SIZE LIMITATIONS FOR UNIFICATION PROBLEMS 33

III AC UNIFICATION TIMES 39

IV UNIFICATION TIMES VS. FORTENBACHER 41

V UNIFICATION TIMES VS. CHRISTIAN & LINCOLN 42

I. INTRODUCTION

A. STRUCTURE

Chapter II contains definitions of terms used throughout the paper. Additional definitions are presented as needed in the remainder of the paper.

In Chapter III, we describe unification, a process which, through substitution, makes two or more terms equal. Applications of unification are discussed and Robinson's algorithm for finding the most general unifier of a set of terms is presented. Finally, we describe E-unification, in which expressions are made equivalent modulo associativity and commutativity.

In Chapter IV, Stickel's AC unification algorithm is examined. This algorithm finds unifiers of functions that are associative and commutative. The algorithm consists of two parts, the first unifies terms in which all of the function's arguments are variables. The second part, which makes use of the first variable-only part, unifies terms in which the arguments can be any term. Chapter IV concludes with a discussion of methods to solve a homogeneous linear Diophantine equation. Stickel's algorithm requires finding a basis of solutions to this type of equation.

In Chapter V, we discuss parallel programming. The two basic types of parallel computing architectures are discussed. Shared memory machines have random-access memory available to several processors, whereas other multiprocessors communicate through message-passing. We discuss considerations and obstacles in developing software capable of exploiting parallel hardware as well as tools designed to facilitate parallel software development.

Chapters VI and VII deal with our sequential and parallel implementations of Stickel's algorithm respectively. In Chapter VI, we discuss the data structures used, the modifications we made to Robinson's and Stickel's algorithms, and various other details of our sequential program. In Chapter VII, we discuss where in our program parallelism could be exploited, the types of messages the processes use to communicate with each other, and how they interact with one another.

In Chapter VIII, we compare the performance of the sequential and parallel programs. We find that our attempt to increase execution speed through distributing the program over a local area network failed. However, we do not see this as an end to our research and suggest further directions the research can take.

Appendix A contains the source code for the parallel implementation of our program. The source for the sequential program is not included since nearly all of it is duplicated in the parallel version.

B. GOALS AND MOTIVATION

In the heart of most automated reasoning systems is a unification algorithm. Every step of reasoning involves a number of unifications to be attempted. An increase in the speed of unification will result in a considerable increase in the speed of the automated reasoning system as a whole.

Unlike standard unification in which two terms have at most one most general unifier, AC-unification may produce a large set of unifiers which may or may not be minimal. These unifiers each correspond to a solution of a homogeneous linear Diophantine equation representing the arguments of the terms being unified and the number of occurrences of these arguments. The unifier corresponding to one solution is independent of any of the other solutions. This appears to be a good opportunity to exploit parallelism. By letting a different processor find the unifiers for a given

solution of the Diophantine equation, the elapsed time needed to find the unifiers should be reduced.

Although no true multiprocessor machines are easily accessible to us at the present time, it is possible to implement a parallel program on a local area network of workstations. We implemented a parallel version of Stickel's AC unification algorithm on six Sun 4/110 workstations interconnected by a CSMA/CD LAN. The processes running on each machine communicate by passing messages to one another across the network.

II. DEFINITIONS

A. TERMS

Variables 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 $i \geq 0$. *Function symbols* are designated by the names $+$, $\times$, f, g, h, f_i, g_i and h_i for $i \geq 0$. *Constants* are designated by the names a, b, c, d, e, a_i, b_i, c_i, d_i and e_i for $i \geq 0$.

A *term* is defined recursively as follows:

(1) A variable is a term.

(2) A constant is a term.

(3) If f is a function symbol and $t_1, \ldots, t_n$ are terms, $f(t_1, \ldots, t_n)$ is a term.

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

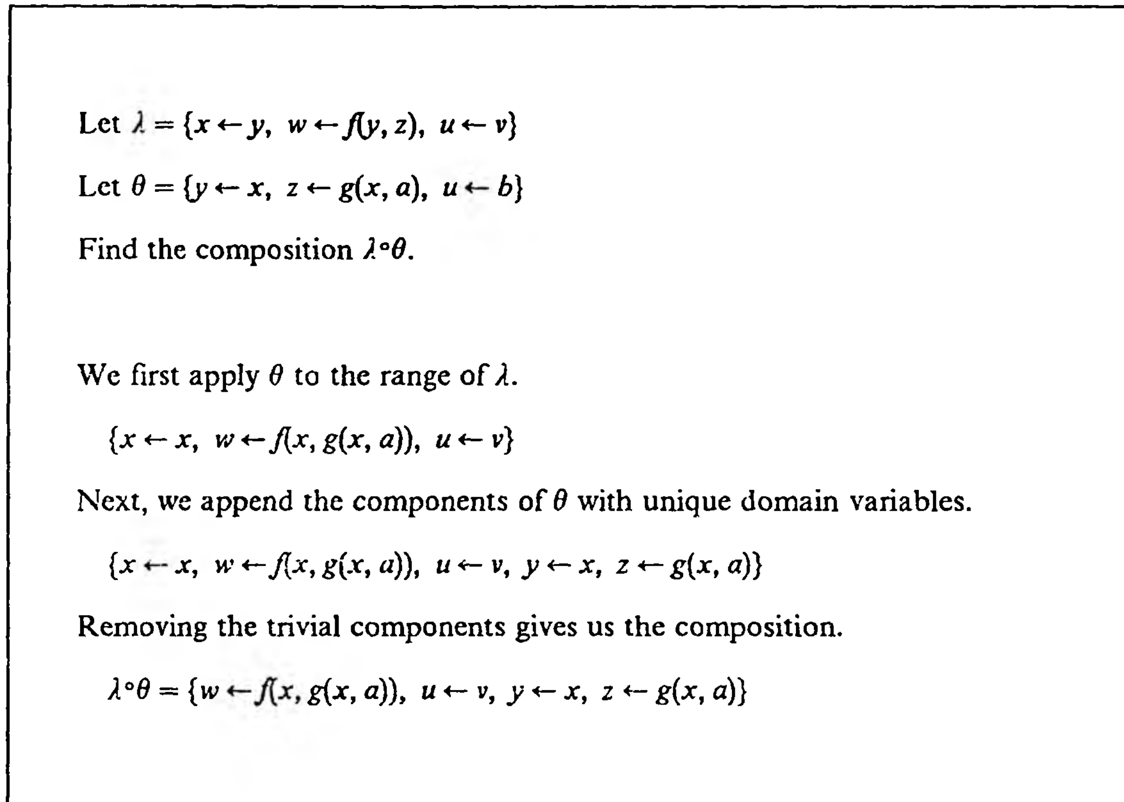
B. SUBSTITUTIONS

A *substitution* represented by the names θ, λ, σ, θ_i, λ_i and σ_i where $i \geq 0$, is a function mapping variables into terms. It is written $\theta = \{v_1 \leftarrow t_1, \ldots, v_n \leftarrow t_n\}$ where $n \geq 0$. Since a substitution is a mapping, the v_i's are distinct such that $v_i \neq v_j$ for $i \neq j$. The empty substitution is represented by ε.

A substitution θ is *applied to* a term t by *simultaneously* replacing every variable in t that is in the domain of θ by the corresponding term. We write $t\theta$ to represent θ applied to t.

For substitutions $\lambda = \{x_1 \leftarrow s_1, \ldots, x_n \leftarrow s_n\}$ and $\theta = \{y_1 \leftarrow t_1, \ldots, y_m \leftarrow t_n\}$, the *composition* of λ and θ, written $\lambda \circ \theta$ is defined $\lambda \circ \theta = \{x_1 \leftarrow s_1\theta, \ldots, x_n \leftarrow s_n\theta\} \cup \{y_i \leftarrow t_i \mid 1 \leq i \leq m,\ y_i \neq x_j \text{ for } 1 \leq j \leq n\}$. To construct the composition, we first apply θ to each term in the range of λ. We then append to λ each component $y_i \leftarrow t_i \in \theta$

such that y_i is not in the domain of λ. If any trivial components of the form $x \leftarrow x$ remain, they can be removed from the resulting substitution. Figure 1 contains an example.



Let $\lambda = \{x \leftarrow y,\ w \leftarrow f(y, z),\ u \leftarrow v\}$

Let $\theta = \{y \leftarrow x,\ z \leftarrow g(x, a),\ u \leftarrow b\}$

Find the composition $\lambda \circ \theta$.

We first apply θ to the range of λ.

$\{x \leftarrow x,\ w \leftarrow f(x, g(x, a)),\ u \leftarrow v\}$

Next, we append the components of θ with unique domain variables.

$\{x \leftarrow x,\ w \leftarrow f(x, g(x, a)),\ u \leftarrow v,\ y \leftarrow x,\ z \leftarrow g(x, a)\}$

Removing the trivial components gives us the composition.

$\lambda \circ \theta = \{w \leftarrow f(x, g(x, a)),\ u \leftarrow v,\ y \leftarrow x,\ z \leftarrow g(x, a)\}$

Figure 1. The Composition of Two Substitutions

The effect of applying the composition $\lambda \circ \theta$ to term t is the same as first applying λ to t and then applying θ to that result. Hence, $t(\lambda \circ \theta) = (t\lambda)\theta$. It is easily shown that for any substitution λ, $\varepsilon \circ \lambda = \lambda \circ \varepsilon = \lambda$.

A term s is an *instance* of a term t, and t is a *generalization* of s if there exists a substitution θ such that $t\theta = s$.

C. PREDICATES AND FIRST-ORDER LOGIC

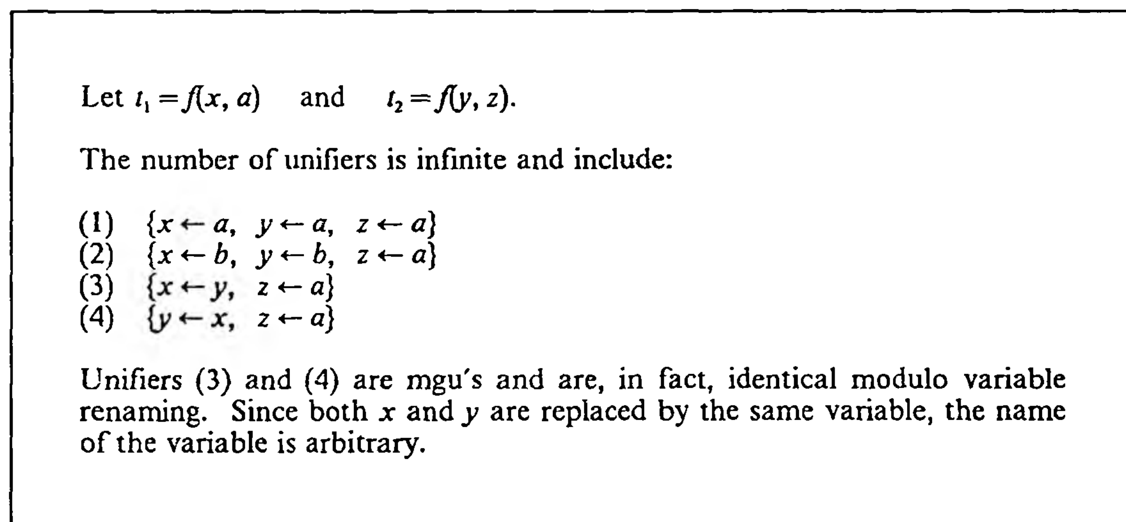
A *predicate* is a constant or function whose range is the set {*TRUE*, *FALSE*}.

A *literal* is either (1) a predicate or (2) the negation of a predicate. A literal of the first form is called a *positive literal* and a literal of the second form is called a *negative literal.*

A *clause* is a disjunction of literals and an *assertion* is a conjunction of literals.

III. UNIFICATION

Unification is a pattern matching process in which two or more terms are made equal by substitutions of their variables. A set of terms is said to be *unifiable* if there exists a substitution which, when applied to each of them, makes them equal. This substitution is called a *unifier.* A unifier μ is called a *most general unifier* or *mgu* of a set of terms if, for every unifier σ of the set, there exists a substitution λ such that $\mu \circ \lambda = \sigma$. Figure 2 gives an example.



Let $t_1 = f(x, a)$ and $t_2 = f(y, z)$.

The number of unifiers is infinite and include:

(1) $\{x \leftarrow a,\ y \leftarrow a,\ z \leftarrow a\}$
(2) $\{x \leftarrow b,\ y \leftarrow b,\ z \leftarrow a\}$
(3) $\{x \leftarrow y,\ z \leftarrow a\}$
(4) $\{y \leftarrow x,\ z \leftarrow a\}$

Unifiers (3) and (4) are mgu's and are, in fact, identical modulo variable renaming. Since both x and y are replaced by the same variable, the name of the variable is arbitrary.

Figure 2. Unifiers and mgu's

A. APPLICATIONS

Unification has applications in several areas including automated theorem proving, expert systems, automated term rewriting systems, and logic programming.

In automated theorem proving systems, expert systems, and logic programming systems, information known to the system is represented as a set of clauses. In order to derive new facts, two or more clauses are combined in such a way that a new clause is formed. The most common way of doing this is using a rule called *resolution*

[$Ro65$]. Using resolution, two clauses, one containing a literal and the other containing its negation, are appended, removing the literal and its negation. The following example illustrates this.

$$\frac{\begin{array}{r} P \mid Q \\ \neg P \mid R \end{array}}{Q \mid R}$$

If the literals appearing in both clauses do not match exactly but are unifiable, we can combine the two clauses in a similar manner, applying the unifier to the other literals in the clauses. This is illustrated below.

$$\frac{\begin{array}{r} f(x) \mid g(x) \\ \neg f(a) \mid h(b) \end{array}}{g(a) \mid h(b)}$$

Here $f(x)$ is unified with $f(a)$ resulting in the unifier $\{x \leftarrow a\}$. This unifier is applied to $g(x)$ and $h(b)$.

Term rewriting, which is also critical to automated theorem proving, is a technique in which terms are replaced with equivalent, and hopefully simpler, terms. For instance, a term rewriting system may replace the term $\cos(x)\tan(x)$ by $\sin(x)$. Rewriting systems contain *rewrite rules* which consist of two terms. Terms are rewritten by *matching* a term with one half of a rewriting rule. Matching is a form of unification in which only the variables of one term, in this case the one in the rewriting rule, can be substituted into. If the terms match, the unifier is applied to the second half of the rewrite rule and the result is a term equivalent to the original one. For example, let $f(x, f(y, z)) \rightarrow f(f(x, y), z)$ be a rewrite rule. Given the term $f(a, f(g(a, b), c))$, we can match the term with the left side of the rewrite rule giving us the substitution $\{x \leftarrow a,\ y \leftarrow g(a, b),\ z \leftarrow c\}$. Applying the unifier to the right side of the rewrite rule, we get $f(f(a, g(a, b)), c)$ which is equivalent to the original term.

Rewrite rules may be applied to subterms imbedded in larger terms. For instance,the term $f(a, +(b,0))$ can be simplified using the rewrite rule $+(x,0) \rightarrow x$ resulting in the term $f(a, b)$.

B. ROBINSON UNIFICATION

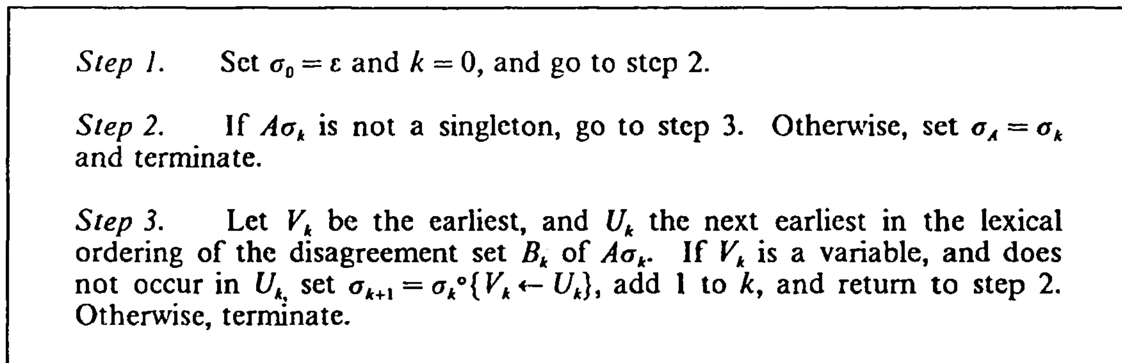
The concept of unification dates back to the introduction of Herbrand's theorem [*He*30]. However, it was not until Robinson [*Ro*65] presented his landmark paper on resolution as a theorem-proving tool that there was an efficient algorithm for finding a unifier. Almost all theorem-proving and term-matching systems to date use Robinson's algorithm or some extension of it.

In order to present Robinson's unification algorithm, we make the assumption that a function symbol is used exclusively for some n-ary function. That is to say, if some function symbol is used to denote a function with 4 arguments, it may not be used to denote a function with 3 arguments elsewhere. We also assume that there is a lexical ordering of all symbols such that the symbols representing variables appear before those representing functions or constants.

We must also introduce an additional definition. Let A be a set of terms. We call B the *disagreement set* of A where B is the set of all subterms of the terms in A which begin at the first symbol position at which not all the terms of A have the same symbol. For example, let $A = \{f(x, g(x,y)), f(x,z), f(x, h(a, b))\}$. The disagreement set of A is $\{g(x,y),\ z,\ h(a,b)\}$. We can see that the disagreement set of A is empty if and only if A is empty or a singleton.

We extend the definition of applying a substitution to a set of terms such that $\{t_1, \ldots, t_n\}\theta = \{t_1\theta, \ldots, t_n\theta\}$. Hence, if θ unifies A, $A\theta$ is a singleton.

Robinson's unification algorithm is shown in figure 3.



Step 1. Set $\sigma_0 = \varepsilon$ and $k = 0$, and go to step 2.

Step 2. If $A\sigma_k$ is not a singleton, go to step 3. Otherwise, set $\sigma_A = \sigma_k$ and terminate.

Step 3. Let V_k be the earliest, and U_k the next earliest in the lexical ordering of the disagreement set B_k of $A\sigma_k$. If V_k is a variable, and does not occur in U_k, set $\sigma_{k+1} = \sigma_k \circ \{V_k \leftarrow U_k\}$, add 1 to k, and return to step 2. Otherwise, terminate.

Figure 3. Robinson's Unification Algorithm

Simply stated, Robinson's algorithm begins with an empty unifier μ. If the set of terms A is a singleton, μ is the most general unifier and the algorithm terminates. Otherwise, it scans the terms until it finds a symbol which is not the same in all the terms. It then considers the subterms beginning at this symbol. It chooses two such subterms. (There may or may not be more.) If one of these subterms V is a variable, it checks if the variable occurs in the second subterm U. If it does, there is no unifier and the algorithm terminates. This is called the occurs check. Otherwise, V is replaced by U in A and μ is composed with $\{V \leftarrow U\}$. If neither subterm is a variable, there is no unifier and the algorithm terminates. If the algorithm has not yet terminated, it repeats with A and μ updated. Figure 4 contains an example.

Robinson's algorithm will always terminate and, if a set of terms is unifiable, it will find the most general unifier. This algorithm, however, could require exponential time due to the occurs check. Several methods have been devised to improve the efficiency of Robinson's algorithm. In fact, the logic programming language PROLOG does not implement the occurs check at all. It is up to the PROLOG programmer to ensure that a term will not be unified with a variable that occurs in

Let $A = \{f(x, g(y, b), g(y, c)), f(g(a, u), u, g(b, c))\}$.

For $k = 0$:

$\sigma_0 = \varepsilon$

$A\sigma_0 = \{f(x, g(y, b), g(y, c)), f(g(a, u), u, g(b, c))\}$

$B_0 = \{x, g(a, u)\}$

For $k = 1$:

$\sigma_1 = \sigma_0 \circ \{x \leftarrow g(a, u)\} = \{x \leftarrow g(a, u)\}$

$A\sigma_1 = \{f(g(a, u), g(y, b), g(y, c)), f(g(a, u), u, g(b, c))\}$

$B_1 = \{g(y, b), u\}$

For $k = 2$:

$\sigma_2 = \sigma_1 \circ \{u \leftarrow g(y, b)\} = \{x \leftarrow g(a, g(y, b)), u \leftarrow g(y, b)\}$

$A\sigma_2 = \{f(g(a, g(y, b)), g(y, b), g(y, c)), f(g(a, g(y, b)), g(y, b), g(b, c))\}$

$B_2 = \{y, b\}$

For $k = 3$:

$\sigma_3 = \sigma_2 \circ \{y \leftarrow b\} = \{x \leftarrow g(a, g(b, b)), u \leftarrow g(b, b), y \leftarrow b\}$

$A\sigma_3 = \{f(g(a, g(b, b)), g(b, b), g(b, c))\}$

Since $A\sigma_3$ is a singleton, σ_3 is the mgu of A.

Figure 4. An example of Robinson's unification algorithm

within the term itself. The trade off, of course, is faster execution time of PROLOG programs.

C. E-UNIFICATION

In theorem-proving and term-matching applications, we often work with functions which have properties such as associativity, commutativity, identity or idempotence. For instance, if the function f is commutative, the terms $f(a, x)$ and $f(b, y)$ will not unify under ordinary unification, although the substitution $\{x \leftarrow b,\ y \leftarrow a\}$ will make the terms equal under the commutative property.

One solution to this problem is to build into the rule base of the system, rewriting rules that will generate every equivalent expression for the terms with regard to the property or properties belonging to each function. Unfortunately, this strategy may generate an excessive amount of useless clauses which will impede the efficiency of the system.

A more elegant solution is to build these properties into the unification algorithm. This method has the advantage of unifying equivalent terms without having to rewrite the terms in a form in which they unify under ordinary resolution. This greatly decreases the number of intermediate clauses used in solving a problem. A disadvantage is that for every property or set of properties, a new algorithm must be developed. For instance, a separate algorithm is needed for functions which are associative only, commutative only, or both associative and commutative. This type of unification is called *E-unification* where E represents the equations or axioms defining the property or properties. Hence, AC-unification is unification under the associative and commutative properties. In the context of E-unification, ordinary Robinson unification is called null-E unification.

Unlike ordinary or null-E unification, E-unification does not guarantee a single mgu. A set of unifiers $\{\theta_1 \ldots \theta_n\}$ is said to be *complete* if for any unifier σ, there exists $1 \leq i \leq n$ and λ such that $\theta_i \circ \lambda = \sigma$. The set of unifiers is *minimal* if for $1 \leq i \leq n$, $1 \leq j \leq n$ and $i \neq j$, there does not exist λ such that $\theta_i \circ \lambda = \theta_j$. It is not uncommon for

a minimal, complete set of unifiers of two relatively short terms to contain hundreds or even thousands of unifiers.

Consider the associative and commutative function f. The minimal, complete set of unifiers for the terms $f(x, a)$ and $f(u, b, v)$ is

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

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

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

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

Here, z_1 and z_2 are variables not in the original terms. These are called ***introduced variables***.

It should be noted that since f is associative, the term $f(u, b, v)$ in the previous example could have been written as $f(f(u, b), v)$ or $f(u, f(b, v))$. Removing nested function symbols in this manner is called *flattening*.

IV. STICKEL'S AC UNIFICATION ALGORITHM

Stickel [*St*81] presented an algorithm for unifying two terms whose function is associative and commutative. Because associative terms can be flattened, we assume the associative and commutative (AC) function can have an arbitrary number of arguments. AC unification is also known as *bag unification* and can be thought of as unifying two multisets since the terms can be flattened and are order-independent.

To unify the terms, they are first flattened and arguments common to both are removed from both terms. Removing the common arguments may eliminate the generation of unifiers that, although correct, are less general than other unifiers. For instance, if the common argument $g(x)$ is not eliminated from the terms, $f(g(x), g(a))$ and $f(g(y), g(x))$ whose most general unifier is $\{y \leftarrow a\}$, unification may result in the additional generation of the unifier $\{x \leftarrow a,\ y \leftarrow a\}$.

Stickel first presents an algorithm for unifying two AC terms whose arguments are all variables. This algorithm, solving the variable-only case, is used by Stickel's general-case algorithm which unifies any two AC terms.

A. THE VARIABLE-ONLY CASE

In the case that all arguments are variables, to unify the terms $f(x_1, \dots, x_n)$ and $f(y_1, \dots, y_m)$ we assign each variable a term of the form t_i whose function symbol is not f or a term of the form $f(t_1, \dots, t_k)$. For such an assignment to be a unifier, each term t_i must appear an equal number of times in each term. Let $s_1 = f(x, x, y)$ and $s_2 = f(u, v, v, w)$. $\theta = \{x \leftarrow a,\ y \leftarrow f(b, b),\ u \leftarrow a,\ v \leftarrow b,\ w \leftarrow a\}$ is a unifier of s_1 and s_2 since $s_1\theta = s_2\theta = f(a, a, b, b)$.

Each term t_i in the substitution must conform to the homogeneous linear Diophantine equation

$$\sum_{j=1}^{m} a_j x_j = \sum_{k=1}^{n} b_k y_k$$

in which a_j and b_k represent the number of occurrences of the jth and k^{th} variable in the first term and second terms, respectively, and x_j and y_k represent the number of times t_i appears in the substitution of the jth and k^{th} variable in the first and second terms, respectively.

For instance, the equation corresponding to the terms s_1 and s_2 is $2x_1 + x_2 = y_1 + 2y_2 + y_3$. Nonnegative integral solutions to this equation can be used to represent unifiers since each variable can be assigned a nonnegative integral number of occurrences of each term. Although the number of solutions to a homogeneous linear Diophantine equation is infinite, we can find a finite set of basis solutions such that each solution is a linear combination of these basis solutions. Table I contains the basis solutions to the above Diophantine equation.

Table I. BASIS OF SOLUTIONS TO DIOPHANTINE EQUATIONS

x	y	u	v	w	
0	1	0	0	1	z_1
0	1	1	0	0	z_2
0	2	0	1	0	z_3
1	0	0	0	2	z_4
1	0	0	1	0	z_5
1	0	1	0	1	z_6
1	0	2	0	0	z_7

Associated with each basis equation is an introduced variable z_i. For each combination of basis equations such that there is at least one nonzero coefficient corresponding to each original variable, we can construct a unifier. The term replacing each variable is made of the introduced variables associated with the basis equations. The coefficient corresponding to an original variable and an introduced variable determines the number of times the introduced variable is represented in the term replacing the original. For instance, the unifier generated from basis equations 3, 4 and 6 is $\{x \leftarrow f(z_4, z_6),\ y \leftarrow f(z_3, z_3),\ u \leftarrow z_6,\ v \leftarrow z_3,\ w \leftarrow f(z_4, z_4, z_6)\}$. The variable-only algorithm is shown more formally in figure 5.

1. Eliminate common terms.

2. Form an equation from the two terms where the coefficient of each variable in the equation is equal to the multiplicity of the corresponding variable in the term.

3. Generate a basis of nonnegative integral solutions to the equation.

4. Associate with each solution a new variable.

5. For each sum of the solutions (no solution occurring in the sum more than once) with no zero components, assemble a unifier composed of assignments to the original variables with as many of each new variable as specified by the solution element in the sum associated with the new variable and the original variable.

Figure 5. Stickel's Variable-Only AC Unification Algorithm

B. GENERAL CASE

To find the unifiers for AC terms with arbitrary arguments (which may be AC functions, ordinary functions, constants or variables) we create two new terms called the *variable abstraction* of the original terms by replacing each distinct argument with a new variable. For instance, the variable abstraction of $f(a, a, x)$ and $f(y, y, b)$ is $f(x_1, x_1, x_2)$ and $f(y_1, y_1, y_2)$ with the substitution $\{x_1 \leftarrow a,\ x_2 \leftarrow x,\ y_1 \leftarrow y,\ y_2 \leftarrow b\}$.

We next use the variable-only algorithm to find the unifiers to the variable abstraction. For efficiency, we introduce additional constraints for generating the variable-only unifiers. Any unifier which assigns a nonvariable to an argument corresponding to a nonvariable in the original terms is eliminated. Likewise any unifier which assigns the same variable to two arguments corresponding to arguments in the original terms that obviously will not unify are discarded. In the above example the unifiers are

(1) $\{x_1 \leftarrow z_4,\ x_2 \leftarrow z_1,\ y_1 \leftarrow z_4,\ y_2 \leftarrow z_1\}$
(2) $\{x_1 \leftarrow z_4,\ x_2 \leftarrow f(z_1, z_2, z_2),\ y_1 \leftarrow f(z_2, z_4),\ y_2 \leftarrow z_1\}$

The last step is to unify each of these unifiers with the substitution corresponding to the variable abstraction. In this example, this is the unifier $\{x_1 \leftarrow a,\ x_2 \leftarrow x,\ y_1 \leftarrow y,\ y_2 \leftarrow b\}$.

This gives us the following results.

(1) $\{x \leftarrow b,\ y \leftarrow a\}$
(2) $\{x \leftarrow f(b, z_2, z_2),\ y \leftarrow f(z_2, a)\}$

Figure 6 contains Stickel's general AC unification algorithm. Note that this algorithm may be called recursively in step 3 for terms with AC functions and that Robinson's unification algorithm is called for all other terms.

1. Form generalizations (the variable abstraction) of the two terms by replacing each distinct argument by a new variable.

2. Use the algorithm for the variable-only case to generate unifiers for the generalizations of the two terms. The variable-only-case algorithm may be constrained to eliminate the generation of unifiers assigning more than one term to variables whose value must be a single term, and the generation of unifiers which will require the later unification of terms which are obviously not unifiable.

3. Unify for each variable in the substitution from step 1 and the unifiers from step 2 the variable values and return the resulting assignments for variables of the original terms. This is the complete set of unifiers of the original terms.

Figure 6. Stickel's General AC Unification Algorithm

C. SOLVING THE DIOPHANTINE EQUATION

Stickel's AC unification algorithm requires a basis of solutions for a homogeneous linear Diophantine equation with integer coefficients of the form

$$\sum_{i=1}^{m} a_i x_i = \sum_{j=1}^{n} b_j y_j .$$

Several algorithms have been developed to generate such a basis.

1. Huet's Algorithm. Huet [*Hu*78] developed an algorithm which begins with the trivial (all zero) solution and generates basis solutions by enumeration. His algorithm determines bounds which provide stopping conditions for the enumeration.

Huet proves that any solution, such that some x_i is greater than the largest b_j or likewise some y_j is greater than some a_i, is nonminimal. Huet also shows that for any $1 \le i \le m$ and $1 \le j \le n$, where lcm_{ij} is the least common multiple of a_i and b_j, a

solution such that $x_i = \mathrm{lcm}_{ij}/a_i$ and $y_j = \mathrm{lcm}_{ij}/b_j$ and all other coordinates 0, is minimal. Therefore, any solution in which all coordinates are greater than or equal to the respective coordinates of any of these solutions is nonminimal. Using these constraints, Huet constructs bounds to limit the enumeration. Each potential solution within these bounds is checked whether it is a solution and that it is not greater than any solution already found.

2. Lankford's Algorithm. Lankford [*La*87] developed an algorithm which uses elementary row operations on a matrix to generate a basis of solutions. By keeping the matrix irredundant, Lankford ensures the basis formed is minimal.

Lankford represents the homogeneous equation as

$$\sum_{i=1}^{m} a_i x_i - \sum_{j=1}^{n} b_j y_j = 0.$$

The *norm* of an $m+n$-tuple S is defined as

$$\|S\| = \sum_{i=1}^{m} a_i s_i - \sum_{j=1}^{n} b_j s_{m+j}.$$

We define A to be the set of all $m+n$-tuples S such that $1 \leq i \leq m$, $s_i = 1$ and all other coordinates are 0. Likewise, B is the set of all $m+n$-tuples S such that $m+1 \leq i \leq m+n$, $s_i = 1$ and all other coordinates are 0.

Lankford's algorithm iteratively finds the sets X^k, P^k, N^k, and Z^k. The initial conditions are

$$X^1 = \text{the empty set},$$
$$P^1 = A,$$
$$N^1 = B,$$
$$Z^1 = \text{the empty set}.$$

The inductive definition of the subsequent generations is

$$X^{k+1} = (A + N^k) \bigcup (B + P^k),$$
$$P^{k+1} = \{S \mid S \in X^{k+1}, \|S\| > 0, \text{ and } S \text{ is irreducible relative to } Z^k\},$$
$$N^{k+1} = \{S \mid S \in X^{k+1}, \|S\| < 0, \text{ and } S \text{ is irreducible relative to } Z^k\},$$
$$Z^{k+1} = Z^k \bigcup \{S \mid S \in X^{k+1} \text{ and } \|S\| = 0\}.$$

In the above definition, S is reducible relative to Z^k if there exists some $Z \in Z^k$ such that each coordinate of X is greater than or equal to the corresponding coordinate of S.

The algorithm terminates when P^k and N^k are empty. When this occurs, Z^k contains an irredundant basis.

3. Zhang's Algorithm. Zhang [*Zh*87] developed a very efficient algorithm which finds the basis solutions to a homogeneous linear Diophantine equations in which several coefficients are 1's. In practice, many of the Diophantine equations appearing in AC unification problems are of this form. The simple case solves Diophantine equations in which all the coefficients on one side of the equation are 1's. This algorithm has the additional asset that intermediate results can be stored and need not be recalculated every time they are needed. The general case solves equations which have two or more 1-coefficients regardless of where they are in the

equation. Zhang's algorithm reduces the equation to smaller equations with only one 1-coefficient and uses some other algorithm (possibly Huet's or Stickel's) to solve them. The smaller equations will generally require less time to solve than the original equation.

Zhang's simple case algorithm considers Diophantine equations of the form

$$\sum_{i=1}^{m} x_i = \sum_{j=1}^{n} b_j y_j .$$

Zhang defines a set $C(m, k) = \{(k_1, \dots, k_m) \mid k_1 + \cdots + k_m = k\}$ and a vector e_n^j which is a vector of length n such that all components are 0 except the jth component is 1. The basis of the Diophantine equation is the set of vectors $X = (x_1, \dots, x_m)$ and $Y = (y_1, \dots, y_n)$ such that $1 \leq j \leq n$, $X \in C(m,j)$, and $Y = e_n^j$. Once $C(m,j)$ is computed, it can be stored and used in solving other equations.

Zhang uses the same concept to simplify finding the basis solutions to Diophantine equations having more than one 1-coefficient, with neither side containing all 1-coefficients. For the details of this algorithm see [Zh87].

V. PARALLEL COMPUTING

Most computer programs today are sequential. They are structured so that one instruction is executed after another in a fixed sequence on a single processor. In recent years, computer hardware technology has made it possible for several processors to be connected in such a way that they can be executing simultaneously and can communicate with each other when necessary. Programs are being designed to distribute their workload so that tasks that are relatively independent of one another's results can be executing at the same time on different processors.

A. PARALLEL HARDWARE

The two major categories of parallel hardware are the *Single Instruction Multiple Data* or SIMD machines and the *Multiple Instruction Multiple Data* or MIMD machines. A SIMD machine consists of a number of processors which execute the same instruction simultaneously on separate data elements. For instance, to add two $m \times n$ matrices, the SIMD machine may execute an add instruction on $m \times n$ different processors, each adding the elements at a different position in the two matrices and storing the result in a third matrix. Of course, if there are less than $m \times n$ processors, the problem is partitioned into smaller pieces and the program would add each piece one after another.

A MIMD machine, on the other hand, can be visualized as several autonomous sequential machines. Each processor can either have its own local memory which it alone can access or processors may share a common memory. A *shared memory* machine has some memory which can be accessed by any of the processors. This scheme allows the processors to communicate through shared variables with almost no overhead. However, care must be taken to ensure that two processors do not attempt to modify the shared data at once.

A *monitor* is a mechanism which allows only one processor to access a section of memory at a single time. A process wanting to access a section of shared memory must first enter the monitor. Once in the monitor, no other process can enter it. When the process is finished accessing the shared memory, it exits the monitor. A process trying to enter a monitor while another process is already in it is put on hold until the other process exits the monitor. To increase efficiency, monitors are made as small as possible so that the probability of two processes needing the monitor at once is minimized.

MIMD machines that do not have shared memory communicate through *message passing*. The processors are connected by some sort of high-speed communications media. They may be directly wired together in a mesh or ring topology or connected to a bus. When a process on one processor wants to send data to a process on another processor, it simply sends a message to that processor. A process wanting data simply receives a message that was sent to it. Obviously, message passing can be implemented on a shared memory machine as well.

A number of workstations on a local area network can be considered a MIMD machine. A single application may run on several workstations, communicating with each other via message passing on the network.

B. PARALLEL SOFTWARE

Although progress is being made rapidly in developing hardware capable of executing parallel programs, software development techniques are making much slower progress. The problem is not that parallel hardware is difficult to use, but that existing applications are usually structured for sequential operation. The actual mechanisms for spawning multiple processes, message passing, and using monitors are usually implemented as primitives that are as easy to use as calling a subroutine.

The difficulty is to partition the program into sections in which the dependency of the results of one section upon the results of any other section is minimized.

Applications vary in the extent to which they can exploit parallelism. Some problems are well suited for parallel execution and easily implemented. Others are very sequential by nature and cannot benefit at all by running in a parallel environment. Many problems have elements which are sequential but can exploit parallelism on a limited basis. Iterative problems in which each iteration uses only the results of past iterations can be broken up into subproblems that can run concurrently, synchronizing when results must be communicated between processes. Some problems consist of a sequence of steps in which some of the steps can exploit parallelism and others cannot. In this case, an application can be running on a single processor part of the time, and on multiple processors another time.

Another consideration for exploiting parallelism is the concept of *grain size*. The grain size refers to amount of work to be done by each process. If a problem is broken up into very small subproblems, the overhead of controlling and communicating with a remote process outweighs the actual work done by the process. On the other hand, if the grain size is too large, the problem may only be broken up into a few pieces and several processors may be idle.

Consider a program that adds two arrays of numbers. If we let each process take a single number from each array and add them, the overhead of setting up the subproblem is much greater than the simple act of adding the numbers. However, if a program finds the prime factors of a list of numbers, and the numbers are sufficiently large, the overhead of sending a single number to a process and later obtaining the results may be significantly less than the act of factoring the number.

There is a lot more to converting a sequential program to run on a parallel machine than recompiling the source on that machine. The programmer must first have a good understanding of the overall program. He must carefully analyze the data transfer within the program and recognize where parallelism can and cannot be exploited. The program will need to be restructured and split into separate executable modules, some of which may be executed on several processors.

Another major obstacle for program development in a parallel environment is debugging. Instead of tracing through the execution of a single program, the programmer or debugger has several concurrent program traces to monitor. A program which may execute flawlessly one time may malfunction the next time because events occur in a different order.

Tools are being developed to help overcome these problems. Trace facilities are being built in to the primitives used to control the parallel processes. These help keep track of the sequence of events taking place in several processes. Windowing systems on workstation consoles allow interactive debugging on more than one process at the same time. By stepping through several processes, the programmer can monitor the interactions between the processes.

C. ARGONNE'S TOOLS FOR PORTABLE PARALLEL PROGRAMS

A set of tools was developed at Argonne National Laboratory to facilitate writing portable C programs to run on a wide range of parallel machines [*BB*87] These tools are implemented as *macros*. A source program containing macros is preprocessed by a macro assembler which expands the macros into C source statements. The resulting file is then compiled and linked as usual. Using this scheme, only the macros themselves need be machine-dependent. Any application programs using the macros will need little or no modification to be compiled and run

on a different machine. The macros also make it easy to develop parallel programs with little knowledge of the machines themselves. The macros support either a shared memory or a message passing environment.

1. The Shared Memory Model. The shared memory model is structured around the monitor. Of course if one process is in the monitor, another cannot enter until the first has left it. The MENTER and MEXIT operations enforce this synchronization. If a process enters a monitor using MENTER, no other process is allowed in until the first process executes the MEXIT operation. Immediately following the MEXIT operation a waiting process may enter the monitor.

Often a process wishes to retrieve data from within a monitor that may not yet be there. Rather than repetitively entering the monitor, checking if the data is there, and exiting, the DELAY and CONTINUE operators are used. Here, the process may enter the monitor, using MENTER, and check the data. If it is already there, it gets the data and exits via MEXIT. Otherwise, it goes into a wait state by executing the DELAY operation. This frees the monitor to be entered by another process. The process depositing the data would execute the CONTINUE operation prior to exiting the monitor. The CONTINUE operation wakes up the dormant process and puts it back in the monitor. The process then receives the data, and exits. If more than one process was to receive the data, the exiting process would execute the CONTINUE operation in case another process was in the DELAY queue. Each process being continued would likewise execute the CONTINUE operation until the DELAY queue is empty.

A process depositing data within a monitor may likewise put itself on hold if the data buffer was full. A process receiving data would then execute the CONTINUE operation when space becomes available in the data buffer. Figure 7 contains pseudocode for implementing a simple monitor that is used to send and receive

messages. Only one message is stored in the monitor at one time. FULL is initially false.

```
SEND (message) {
   MENTER(monitor)
   if FULL is true
      DELAY (SENDQ)
   move message to BUFFER
   set FULL = true
   CONTINUE (RECEIVEQ)
   MEXIT(monitor)
}

RECEIVE (message) {
   MENTER(monitor)
   if FULL is false
      DELAY(RECEIVEQ)
   move BUFFER to message
   set FULL = false
   CONTINUE(SENDQ)
   MEXIT(monitor)
}
```

Figure 7. A Monitor For Sending and Receiving Messages

One type of monitor is called a *barrier*. In a barrier, processes entering the monitor enter a delay state until a certain number of processes have entered. Then they are released one after another. The barrier allows several tasks to synchronize. Barriers are often used at the end of some problem to insure that no process continues until they all have finished.

Argonne's macros include higher level monitors which relieve the programmer from handling the mechanics of scheduling jobs to several processors. The GETSUB monitor assigns subscripts sequentially to each process entering it. For instance, if every element of an array is processed independently, each processor retrieves the subscript of an array element, processes that element, and gets another subscript as

soon as it finishes. The monitor insures that two processes do not get the same subscript.

Another higher level monitor is the ASKFOR monitor. The ASKFOR monitor manages a pool of tasks which are distributed to processes when they enter the monitor. By entering the monitor, the process *asks for* a task. In processing the task, the process may create new subtasks to add to the pool. This procedure can terminate in more than one way. If the pool of tasks is exhausted, there are no problems left to solve. For some problems, once an answer has been found, the rest of the problem can be abandoned. The ASKFOR monitor communicates these conditions to the processes as well as distributing the workload.

2. The Message Passing Model. The message passing model supports parallel programming in an environment in which processes communicate solely through messages. This allows the same applications to be run on diverse machines such as uniprocessor systems, shared-memory multiprocessors, multiprocessors without shared memory, and networks of workstations.

A program under this model consists of three components: the program for the master process, the program(s) for the slave processes, and a table describing how to create the slave processes. The master process uses the table to spawn the slave processes which may be on different machines. The master and slave processes communicate through messages that may contain both control signals and data. When the program is ready to terminate, an end signal is transmitted to all the processes telling them to terminate.

Messages are sent and received by the SEND and RECEIVE macros respectively. The SEND macro has as arguments the receiver's ID, the message type, data, and an optional length. The length will default to the declared length of the

message type. SENDR is an optional form of the SEND macro in that the sender does not procede until the message is acknowledged by the receiver. This alleviates the need for an explicit acknowledge to be designed into the program. The RECEIVE macro indicates to the receiver the type of message received and who sent the message, along with any data in the message. If the message was sent using the SENDR macro, it automatically acknowledges the message. The receiving program can selectively receive a message from a particular sender and/or receive a particular type of message.

The messages transmitted must be explicitly typed using the MSG_TYPE macro. This way, if messages are passed between two machines which represent data differently, the type conversion is transparent to the programmer. The SEND, SENDR, and RECEIVE macros handle the data conversion so that, for instance, a double precision floating point number is represented correctly on both machines. The data type is represented either as EMPTY which specifies there is no data field in the message or as a structure of C data types. Data types such as pointers and unions are not supported as message data types.

Additional macros needed for the message passing program are ENV, INITENV, and WAIT_FOR_END. ENV defines data types and variables used by the other macros. INITENV initializes these variables. WAIT_FOR_END is a macro used by the master process before terminating. It insures that all the slave processes have terminated. Some machines do not allow a process to exit until all processes spawned from it are killed.

VI. SEQUENTIAL IMPLEMENTATION

We first implemented Stickel's AC unification algorithm in C to run sequentially on a single Sun 4/110 workstation. Zhang's simple-case algorithm is used to construct the basis solutions to the Diophantine equations when the coefficients on one side of the equation are all 1's. Otherwise, Lankford's algorithm is used.

A. DATA STRUCTURES

Terms are represented as trees where each node is a function symbol, constant, or variable. Only function symbols have children. The nodes are represented by type (function, constant, or variable) and an integer identifier. The identifiers are associated with character string names by a table. The names are used only for the user interface. Each node has a pointer to its first child, if any, and its next sibling.

A flattened term is represented as a linked list of pointers to the term's arguments. The AC function symbol is not a part of the data structure. This data structure allows the term to appear flat without changing any of the pointers in the term or having to make a copy of the subterms.

Unifiers are represented as linked lists, where each list element consists of a variable identifier and a pointer to a term. A set of unifiers is a linked list of pointers to unifiers.

Substitutions are not made directly to the terms. Instead, whenever a node is inspected, and the node represents a variable, the node's identifier is compared to the variable identifiers in the substitution. If that identifier is found, the term pointed to by the substitution is used in place of the variable If this new term is itself a variable, this process is repeated until either a nonvariable or a variable not in the substitution is reached. If the substitutions were applied to the terms, the terms

could grow exponentially in size as multiple occurrences of variables are replaced with terms which may themselves contain repeated variables requiring substitution.

B. ROBINSON'S UNIFICATION ALGORITHM

Robinson's algorithm had to be modified slightly to work in conjunction with AC functions. It is implemented recursively such that if the two terms to be unified are the same function and have the same number of arguments, the arguments are unified by a generic unification routine. This generic routine determines if the terms are the same AC function and, if they are, it uses Stickel's algorithm, otherwise it uses Robinson's.

Robinson's algorithm was also modified to handle more than one unifier. Since the arguments of the terms may contain AC functions, more than one substitution may be returned by the generic unification routine. Subsequent arguments are unified independently for each of these substitutions. Substitutions resulting from these unifications are accumulated and likewise used in unifying subsequent arguments. Hence, this algorithm may return multiple unifiers.

To prevent the occurs check from redundantly searching the same subterms for an occurrence of the same variable, a list is maintained of variables whose substitutions do not contain the target variable. The first time a variable is encountered that is in the substitution list, the associated term is searched. If the target variable is not found, the encountered variable is put on the list of searched variables. Otherwise, the occurs check fails. If a variable is encountered that is on the list of searched variables, it is immediately skipped.

C. STICKEL'S ALGORITHM

Our implementation of of Stickel's algorithm varies from that presented in Chapter 5. Rather than determining the complete set of unifiers for the variable abstraction and then unifying these with the substitution defining the variable abstraction, we create each variable-only unifier as we need it and unify it with the variable-abstraction substitution. It should be apparent that the variables representing the variable abstraction need not be used in either scheme. It is only the terms associated with these variables that are actually unified.

As in our implementation of Robinson's algorithm, the generic unification function is called to unify the arguments. The existence of multiple unifiers for these subterms is handled in a similar manner.

D. OTHER CONSIDERATIONS

The user interface reads from the UNIX standard input and writes to standard output. The input is read in stream mode such that white space (spaces, tabs, or newline characters) separate the the input symbols. Terms are read two at a time to be unified. Functions are enclosed in parentheses where the first symbol inside is the function symbol and the following terms are arguments. Symbols beginning with the letters u through z are treated as variables. Variables consisting of the letter z followed by an integer such as z12 should be avoided since that is the format of the introduced variables created by the AC unification algorithm.

The program will print the number of unifiers found, if any, and will optionally print up to 100 of them. Also output is the elapsed wall time for functions such as reading the input data, unifying the terms, and printing the unifiers. The wall time is used, rather than cpu time, since it is more meaningful in comparison to the

execution time of a parallel implementation. The wall time is determined by the Berkeley-UNIX C function *gettimeofday*.

Table II contains some size limitations built into the current implementation. Other limits, such as the maximum number of unifiers that can be found, are limited by the amount of memory that can be dynamically allocated by the program.

Table II. SIZE LIMITATIONS FOR UNIFICATION PROBLEMS

Limitation	Value
Max. number of basis solns. to Diophantine eq.	200
Max. number of unique arguments in 2 terms being unified	12
Max. length of symbols	9 chars.
Max. number of unique symbols in two terms being unified	100
Max. number of matrix rows generated for N or P matrices in one step of Lankford's algorithm	25

VII. PARALLEL IMPLEMENTATION

We implemented the parallel version of the AC unification algorithm to run on six Sun 4/110 workstations connected via a CSMA/CD LAN. The master process runs on one workstation and a slave runs on each of the other five. The processes communicate via message passing using the macros developed by Argonne Laboratory.

A. OPPORTUNITIES FOR EXPLOITING PARALLELISM

Our unification program presents several possible opportunities to exploit parallel processing. For instance, the occurs check in Robinson's unification algorithm need not be sequential. Recall that the occurs check checks for an occurrence of some variable in a term. If the term being checked is a function with several arguments, each argument can be checked for the variable by a different processor. However, the overhead of message passing on a LAN far outweighs the processing needed to perform the occurs check on one term. Hence, the grain size of the occurs check is too small to make efficient use of parallelism.

The algorithms used to find the basis solutions to the Diophantine equations similarly can be designed to distribute the processing, but again, the grain size of the problem is rather small compared to the overhead inherent in our message-passing scheme.

Since our implementation of Robinson's algorithm allows multiple substitutions to be returned when unifying a terms arguments, subsequent arguments must be unified using each of the previous substitutions. Each of these unifications may be done by a different processor. Consider the terms $f(+(x,y),+(x,z))$ and $f(+(a,u),+(b,v))$, where $+$ is AC and f is not. Robinson's unification algorithm

would first unify $+(x,y)$ and $+(a,u)$ using the AC unification algorithm. Stickel's algorithm will return four substitutions. The next pair of arguments are unified four times, once for each of these substitutions. These unifications may be each be distributed to a separate processor.

Parallelism can be exploited similarly in the last step of Stickel's algorithm when the subterms corresponding to the variable abstraction are unified with the introduced-variable terms.

Another way to exploit parallelism in Stickel's algorithm is to generate the unifiers corresponding to each particular solution of the Diophantine equation on a separate processor. This method is appealing since the generation of the unifier involves unifying all the arguments of the terms. The grain size of the distributed subtasks is larger than in the previous methods in which each distributed subtask unified one argument.

We decided to implement our program using this last plan. Our AC unification algorithm runs sequentially up to the point where the valid solutions to the Diophantine equation are determined. The master process then sends a solution to each slave, which finds the unifiers associated with that solution.

B. IMPLEMENTATION

The master process contains all of the elements of our sequential implementation with the exception of the sequential AC unification algorithm itself. The user interface and Robinson unification algorithm remain unchanged. The mainline was modified to create and terminate the slave processes.

The slave processes consist of the distributed part of the AC unification algorithm and all routines necessary to unify the subterms. Since the slaves do not have the ability drive other slaves, the sequential version of the AC unification algorithm is included in the slaves.

The processes exchange the following types of messages.

TERMS - sent from the master to the slaves. Contains the terms and a substitution representing the variable abstraction of the arguments of the terms being unified and the substitution calculated so far.

READY - sent from a slave to the master. Signals that the slave is ready to be sent a problem.

PROBLEM - sent from the master to a slave. Contains the basis vectors that make up a particular solution to the Diophantine equation.

SOLUTION - sent from a slave to the master. Contains unifiers corresponding to the problem sent to the slave. A flag indicates if there are more unifiers than can be transmitted in one message.

SEND_MORE - sent from the master to a slave when the slave indicated it had more solutions to send.

DONE - sent by both the master and the slaves. Tells the slave that no more problems will be sent associated with the last terms sent. The slave returns this message in acknowledgement.

END_SIGNAL - sent by both the master and the slaves. Tells the slaves to terminate. The slave returns the message in acknowledgement.

ERROR_SIGNAL - sent by a slave to the master. Signals that a slave cannot continue due to array overflow or insufficient memory. Allows graceful abnormal termination. The master acknowledges by sending END_SIGNAL to all slaves and terminating.

Figures 8 and 9 contain pseudocode for the master and slave components of the AC unification algorithm.

```
AC_unify {
    Flatten both terms.
    Remove arguments common to both terms.
    For each distinct argument {
        Add it to variable abstraction.
        Determine its multiplicity.
    }
    Form Diophantine equation from multiplicities of arguments.
    Find basis of solutions to Diophantine equation.
    Remove illegal basis vectors.
    Determine all valid solutions to Diophantine equation.
    If no solutions exist
        Return(fail).
    Send TERMS to slaves.
    UNIFIERS = empty list.
    NUM_DONE = 0.
    While (NUM_DONE < number of slaves) {
        Receive message from slave.
        If (message = SOLUTION) {
            Add solution to UNIFIERS.
            If (slave has more to send)
                Send SEND_MORE to slave.
        }
        If (message = READY or
            (message = SOLUTION and slave has no more)) {
            If (more problems to send)
                Send PROBLEM to slave.
            Else
                Send DONE to slave.
        }
        Else if (message = DONE)
            NUM_DONE = NUM_DONE + 1.
    }
    If (UNIFIERS = empty list)
        Return(fail).
    ELSE
        Return(success).
}
```

Figure 8. AC Unification Algorithm - Master

```
slave {
   message = READY.
   While (message != END_SIGNAL) {
      Receive message from master.
      If (message = TERMS) {
         Parse terms from message.
         Parse SUBSTITUTION from message.
         Send READY to master.
         message = PROBLEM
         While (message = PROBLEM or message = SEND_MORE) {
            Receive message from master.
            If (message = PROBLEM) {
               Determine variable-only unifier.
               OLD_UNIF = SUBSTITUTION.
               For each term in variable abstraction {
                  NEW_UNIF = empty.
                  For each substitution in OLD_UNIF {
                     Unify variable-only, variable-abstraction terms.
                     Append unifier (if any) to NEW_UNIF.
                  }
                  OLD_UNIF = NEW_UNIF
               }
               While (Too many unifiers for one message) {
                  Send SOLUTION to master.
                  Receive SEND_MORE from master.
               }
               Send SOLUTION to master.
            }
            Else if (message = DONE)
               Send DONE to master.
         }
      }
   }
   Send END_SIGNAL
}
```

Figure 9. AC Unification Algorithm - Slave

VIII. RESULTS

A. OBSERVATIONS

The results of our research are disappointing. In most cases, the algorithm runs faster sequentially than it does in parallel. Table III shows the running time comparison between the two implementations in seconds of wall time. The functions + and * are both associative and commutative. The time of 0 seconds is a result of the granularity of the *gettimeofday* function on the Sun workstations which returns time in increments of 10 ms.

Table III. AC UNIFICATION TIMES

Terms being unified	# sols	seq time	par time
+(x,a,b) +(u,c,d,e)	2	.00	.08
+(x,x,y) +(u,v,v,c)	18	.01	.20
+(x,a,b) +(u,v,c,d)	12	.02	.17
+(x,y,z) +(u,v,w,xx)	2161	2.54	20.75
+(*(a,a,x,x),*(b,c,y,y,z),*(a,b,c,x)) +(*(a,b,u),*(c,c,u,u),*(c,u,v))	51	.33	.64
+(x,*(x,y),*(y,z)) +(*(u,v),*(v,v,a),u)	1610	4.41	23.04
+(*(a,b,x),*(x,y,c),*(x,y,d)) +(*(d,e,f),*(c,u,e),*(v,b,g))	0	.02	.11
+(*(a,x,y),*(b,xx,xy),*(c,yx,yy)) +(*(d,e,f),*(e,uu,vv),*(d,u,v))	0	.01	.10
+(*(a,x,y),*(b,xx,xy),*(c,yx,yy)) +(*(d,u,v),*(d,e,f),*(e,uu,vv))	0	.79	.69
+(*(a,x,y),*(b,xx,yy),*(c,yx,yy)) +(*(d,u,v),*(e,uu,vv),*(d,e,f))	0	42.10	31.18

We would expect the sequential program to perform better on the very simple unifications, due to the fact that the overhead involved in distributing the problem is greater than the unification itself. We see that this is indeed the case. However, we find that the parallel program performs much slower on larger problems in which many unifiers are produced. On large problems that produce few or no unifiers, the parallel program does run faster.

This behavior has led us to the conclusion that even though the unifiers might be calculated by the slaves quickly, the overhead of the master process receiving the data from the slaves is prohibitive. A single processor can find the unifiers using the sequential program faster than it can get them from the slave processes via message passing on the LAN.

We verified this notion by imbedding system clock calls within the master and slave process to determine where the program was spending the most time. The slave processes were found to be unifying the subterms and sending the results to the master process quickly. They spent most of their time waiting for the master process to send more problems. The master process spent most of its time receiving the unifiers from the slaves. It was not possible to tell what part of that time it was idle, waiting for a response from the slaves, but since the slaves *were* spending most of their time waiting, we suspect the master was not.

B. ANALYSIS OF SEQUENTIAL PROGRAM

Since the our parallel implementation of the AC unification algorithm is derived from our sequential implementation, it is critical that the sequential version be as efficient as possible. In tables IV and V, we compare the unification times of our sequential program to those published by Fortenbacher [*Fo*87] and Christian and Lincoln [*CL*87]. Fortenbacher's program was run on an IBM 3081 under CMS and

Christian and Lincoln's program was run on a Symbolics 3600. Again the running times of our program are in wall time rather than cpu time and we are limited to an accuracy of ± 10 milliseconds. Since our program was run on a different machine, we cannot say with certainty how it compares with the other programs. However, the unification times are close enough that we believe our program is an efficient rendering of the algorithm.

Table IV. UNIFICATION TIMES VS. FORTENBACHER

Terms being unified	# sols	Our time	F.'s time
+(a,x) +(b,y)	2	.010	.005
+(x,x) +(y,z)	5	.000	.012
+(x,x,y,a,c) +(b,b,z,c)	4	.000	.012
*(+(x,a),+(y,a),c,c) *(+(z,z,z),w)	4	.010	.030
*(+(x,a),+(y,a),+(z,a)) *(+(w,w,w),z,z)	2	.010	.019
+(x,a,x,a) *(g(a),g(y),z,z)	2	.000	.009
+(w,+(x,g(x))) +(y,+(z,g(z)))	35	.050	.108

C. TOPICS FOR FUTURE RESEARCH

As stated in the previous section, the message-passing overhead on the LAN prohibits us from taking advantage of the computing power available to us on the network. By implementing the parallel program on a machine with shared memory or very fast interprocessor communication, we should be able to increase the performance of the algorithm.

Table V. UNIFICATION TIMES VS. CHRISTIAN & LINCOLN

Terms being unified	# sols	Our time	C & L's time
+(x,a,b) +(u,c,d,e)	2	.000	.005
+(x,a,b) +(u,c,c,d)	2	.010	.005
+(x,a,b) +(u,c,c,c)	2	.000	.004
+(x,a,b) +(u,v,c,d)	12	.020	.013
+(x,a,b) +(u,v,c,c)	12	.010	.014
+(x,a,b) +(u,v,w,c)	30	.050	.034
+(x,a,b) +(u,v,w,ww)	56	.080	.079
+(x,a,a) +(u,c,d,e)	2	.000	.005
+(x,a,a) +(u,c,c,d)	2	.010	.004
+(x,a,a) +(u,c,c,c)	2	.000	.005
+(x,a,a) +(u,v,c,d)	8	.010	.010
+(x,a,a) +(u,v,c,c)	8	.000	.011
+(x,a,a) +(u,v,w,c)	18	.020	.023
+(x,a,a) +(u,v,w,ww)	32	.030	.051
+(x,y,a) +(u,c,d,e)	28	.030	.024
+(x,y,a) +(u,c,c,d)	20	.020	.018
+(x,y,a) +(u,c,c,c)	12	.010	.013
+(x,y,a) +(u,v,c,d)	88	.110	.064
+(x,y,a) +(u,v,c,c)	64	.050	.048
+(x,y,a) +(u,v,w,c)	204	.240	.160
+(x,y,a) +(u,v,w,ww)	416	.510	.402
+(x,y,z) +(u,c,d,e)	120	.120	.118
+(x,y,z) +(u,c,c,d)	75	.060	.072
+(x,y,z) +(u,c,c,c)	37	.030	.038
+(x,y,z) +(u,v,c,d)	336	.340	.269
+(x,y,z) +(u,v,c,c)	216	.170	.171
+(x,y,z) +(u,v,w,c)	870	1.000	.729
+(x,y,z) +(u,v,w,ww)	2161	2.630	1.994

Providing we get over the data transmission obstacle, we may further increase performance by breaking the program up into finer components. Chapter VII discusses several opportunities for exploiting parallelism that we did not attempt. Since we were only working with a small number of processors, it would have done us no good to have the slave processes distribute any of their workload. However, if more processors were available, a slave process might break up its subtask into

smaller subtasks which would be distributed by an ASKFOR monitor as described in chapter V.

On a shared memory machine, the overhead of distributing tasks may be low enough to exploit parallelism in the occurs check or in solving the Diophantine equations. We may also be able to move our data structures into shared memory and use monitors to coordinate communication between the processes, avoiding the overhead of message passing.

REFERENCES

[*BB*87] Boyle J., Butler R., Disz T., Glickfield B., Lusk E., Overbeek R., Patterson J., and Stevers R., *Portable Programs for Parallel Processors*, Holt, Rhinehart and Winston, Inc., New York, 1987.

[*CL*87] Christian J., and Lincoln P., "Adventures in associative-commutative unification." MCC Technical Report ACA-ST-275-87, 1987.

[*Fo*87] Fortenbacher A., "An algebraic approach to unification under associativity and commutativity." *Journal of Symbolic Computation*, vol 3, (1987) pp. 217-229.

[*He*30] Herbrand J., "Recherdies sur la Theorie de la Demonstration." *Trauaux de la Societe des Sciences et Lettres de Varsovie, Classe III Sci. Math. Phys.*, 33, 1930.

[*Hu*78] Huet G., "An algorithm to generate the basis of solutions to homogeneous linear Diophantine equations." *Information Processing Letters*, vol 7, no 3, (April, 1978) pp. 144-147.

[*La*87] Lankford D.S., "Non-negative integer basis algorithms for linear equations with integer coefficients." (unpublished), 1987.

[*Ma*88] Mayfield B., "The role of term symmetry in equational unification and completion procedures." Ph.D. dissertation, University of Missouri-Rolla, 1988.

[*Ro*65] Robinson J.A., "A machine-oriented logic based on the resolution principle." *Journal of the Association for Computing Machinery*, vol 12, no 1, (January, 1965), pp. 23-41.

[*St*81] Stickel M., "A unification algorithm for associative-commutative functions." *Journal of the Association for Computing Machinery*, vol 28, no 3, (July, 1981), pp. 423-434.

[*Wi*88] Wilkerson R., Private communication. 1988.

[*Zh*87] Zhang H., "An efficient algorithm for simple Diophantine equations." Technical Report 87-26, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, New York. 1987.

VITA

David John Kleikamp was born on March 7, 1964 in Lebanon, Missouri. He graduated from Lee's Summit High School in Lee's Summit, Missouri in May, 1982.

He received a Bachelor of Science degree in Computer Science from the University of Missouri-Rolla in Rolla, Missouri in December 1986. As an undergraduate, he was an active member of Sigma Pi Fraternity and worked part time as a student programmer for the United States Geological Survey and as a bartender.

He remained at the University of Missouri-Rolla where he is currently pursuing a Master of Science degree in computer science. He continued working for the U.S. Geological Survey until August 1988 when he began working as a graduate research assistant in the computer science department.

He is engaged to marry Donna Marie Talleur in October 1989.

APPENDIX

SOURCE LISTING - PARALLEL VERSION

The following is an include file defining parameters and data structures used throughout the code.

```
#ifndef TYPES_DEFINED

#define TYPES_DEFINED

#include <stdio.h>
#include <sys/time.h>

#define LOGICAL int
#define TRUE 1
#define FALSE 0

#define NUMAC 2
#define MAXCNO 12
#define NMAXDIMENSION 50
#define ZMAXDIMENSION 200
#define TMAXLEN 1000
#define UMAXLEN 800

#define STAT_TYPE long
#define MAX_CLOCKS 100
#define INPUT_TIME 0
#define UNIFY_TIME 1
#define CLEANUP_TIME 2
#define TOTAL_TIME 3
#define PRINT_TIME 4

#define NSLAVES 5

#define TERMS 0
#define PROBLEM 1
#define SOLUTION 2
#define DONE 3
#define READY 4
#define END_SIGNAL 5
#define ERROR_SIGNAL 6
#define SEND_MORE 7

struct term {
  int id;
  char type;
```

```
    struct term *child;
    struct term *sibling;
};

struct flat {
    struct term *subterm;
    struct flat *next;
};

struct substitution {
    int var;
    struct term *subst;
    struct substitution *nexts;
};

struct unifier {
    struct substitution *first;
    struct unifier *nextu;
};

struct vector {
    int vect;
    struct vector *nextv;
};

struct dio_soln {
    struct vector *first;
    struct dio_soln *next;
};

struct term_list {
    char type;
    int id;
};

struct clock {
    STAT_TYPE accum_sec;   /* accumulated seconds */
    STAT_TYPE accum_usec;  /* accumulated microseconds*/
    STAT_TYPE curr_sec;
    STAT_TYPE curr_usec;
};

#endif
```

The following is the master main-line program. It contains macro calls for the message-passing primitives.

```
#include "types.h"

/******
 *
 *  These macros define the message-passing environment
 *
 ******/
ENV

PROC_ID process_ids[NSLAVES];

BEGIN_MSG_TYPES
  MSG_TYPE(TERMS, struct terms {
  int ac_op;
  int base;
  int nterms;
  struct term_list list[TMAXLEN];
}; )
  MSG_TYPE(PROBLEM, struct problem {
  int nvects;
  int dsolution[ZMAXDIMENSION][MAXCNO];
}; )
  MSG_TYPE(SOLUTION, struct solution {
  int base;
  int nunifiers;
  LOGICAL more;
  struct term_list list[UMAXLEN];
}; )
  MSG_TYPE(SEND_MORE, EMPTY)
  MSG_TYPE(DONE, EMPTY)
  MSG_TYPE(END_SIGNAL, EMPTY)
  MSG_TYPE(READY, EMPTY)
  MSG_TYPE(ERROR_SIGNAL, EMPTY)
END_MSG_TYPES

/******
 *
 *  main is the mainline master process.  It spawns the slave processes and
 *  drives the unification.
 *
 ******/
main(argc, argv)
int argc;
char *argv;
{
  struct term *term1, *term2;
  char string[10];
  struct unifier *unifiers, *u;
  int unify(), count, print_unifiers(), base, i, done;
```

```
    char symtab[100][10];
    int nsymbols, parse_term(), ttime, utime, itime, ctime, ptime;
    void erase_term(), free_unifiers();
    char *alloc(), print_flag;
    STAT_TYPE clock_val();
    PROC_ID id;
    int msg_type;

    INITENV

    clock_init();
    clock_start(TOTAL_TIME);

#if NSLAVES == 1
    REMOTE_CREATE(one_slave, process_ids)
#endif

#if NSLAVES == 3
    REMOTE_CREATE(three_slaves, process_ids)
#endif

#if NSLAVES == 5
    REMOTE_CREATE(five_slaves, process_ids)
#endif

    done = 0;

    printf("\nPrint unifiers? ");
    scanf("%s", string);
    print_flag = string[0];

    while (!done) {
       clock_reset(INPUT_TIME);
       clock_reset(UNIFY_TIME);
       clock_reset(PRINT_TIME);
       clock_reset(CLEANUP_TIME);

       clock_start(INPUT_TIME);

       symtab[0][0] = '+';
       symtab[0][1] = '\0';
       symtab[1][0] = '*';
       symtab[1][1] = '\0';

       nsymbols = 2;

       printf("\nEnter a term: ");

       if (scanf("%s",string) != EOF) {
          if (string[0] != '(' || string[1] != '\0') {
             printf("\nInvalid syntax!\n\n");
             stop();
          }

          term1 = (struct term *) alloc(sizeof(struct term));
```

```
        if (parse_term(term1, symtab, &nsymbols))
            stop();
        printf("\n\nTerm1: ");
        print_term(term1, NULL, symtab);

        printf("\n\nEnter second term: ");
        scanf ("%s",string);

        term2 = (struct term *) alloc(sizeof(struct term));
        if (parse_term(term2, symtab, &nsymbols))
            stop();
        printf("\n\nTerm2: ");
        print_term(term2, NULL, symtab);

        clock_stop(INPUT_TIME);
        clock_start(UNIFY_TIME);

        unifiers = (struct unifier *) alloc(sizeof(struct unifier));
        unifiers->first = NULL;
        unifiers->nextu = NULL;
        base = 1;
        if (unify(term1, term2, &base, NULL, unifiers)) {
            clock_stop(UNIFY_TIME);
            clock_start(PRINT_TIME);
            if (print_flag == 'y') {
                printf("\n\nUnifiers:\n");
                count = print_unifiers(unifiers, symtab);
            }
            else
                for (u = unifiers, count = 0; u != NULL; u = u->nextu, count++);
            printf("\n\nNumber of unifiers = %d", count);
            clock_stop(PRINT_TIME);
        }
        else {
            clock_stop(UNIFY_TIME);
            printf("\nterms will not unify.");
        }

        clock_start(CLEANUP_TIME);
        erase_term(term1);
        erase_term(term2);
        free_unifiers(unifiers);
        clock_stop(CLEANUP_TIME);

        itime = clock_val(INPUT_TIME);
        utime = clock_val(UNIFY_TIME);
        ptime = clock_val(PRINT_TIME);
        ctime = clock_val(CLEANUP_TIME);
        printf("\n\nInput time: %d ms\nUnify time: %d ms\n",itime, utime);
        printf("Print time: %d ms\nCleanup time: %d ms\n", ptime, ctime);
    }
    else
        done = 1;
}
for (i=0; i<NSLAVES; i++)
```

```
    SEND(&process_ids[i], END_SIGNAL);
  for (i=0; i<NSLAVES; i++)
    RECEIVE(&id, &msg_type, , (match_id(&process_ids[i]) &&
                              match_type(END_SIGNAL)))
  WAIT_FOR_END(NSLAVES)
  ttime = clock_val(TOTAL_TIME);
  printf("\n\nTotal time: %d ms\n", ttime);
}
```

The following is AC unification algorithm used by the master process. It includes the macro calls to invoke the message-passing primitives.

```
#include "types.h"
/********
 *
 * These macros must be included to define the message passing environment
 *
 ********/
ENV

BEGIN_MSG_TYPES
  MSG_TYPE(TERMS, struct terms {
  int ac_op;
  int base;
  int nterms;
  struct term_list list[TMAXLEN];
}; )
  MSG_TYPE(PROBLEM, struct problem {
  int nvects;
  int dsolution[ZMAXDIMENSION][MAXCNO];
}; )
  MSG_TYPE(SOLUTION, struct solution {
  int base;
  int nunifiers;
  LOGICAL more;
  struct term_list list[5000];
}; )
  MSG_TYPE(SEND_MORE, EMPTY)
  MSG_TYPE(DONE, EMPTY)
  MSG_TYPE(END_SIGNAL, EMPTY)
  MSG_TYPE(READY, EMPTY)
  MSG_TYPE(ERROR_SIGNAL, EMPTY)
END_MSG_TYPES

extern PROC_ID process_ids[NSLAVES];

/******
 *
 * AC_unify is the parallel version of Stickel's algorithm.
 *
 ******/
int AC_unify(term1, term2, base, old_unifier, new_unifiers)
struct term *term1, *term2;
int *base;
struct substitution *old_unifier;
struct unifier *new_unifiers;
{
  void elim_dup();
  struct flat *new1, *new2, *flatten(), *ptr2, *ptr;
  struct term *arg, *arg2, *last, *parse_t();
  struct substitution *copy_s(), *unif;
```

```
struct vector *v, *tempv;
struct unifier *new, *final;
int n1, n2, nA, nB, same_term(), count_subterms();
int find_basis(), i, j;
int npairs, num, basis[ZMAXDIMENSION][MAXCNO+1];
int weed_basis(), highest, offset;
struct dio_soln *solution, *sol, *find_solutions(), *tempsol;
int so_far[ZMAXDIMENSION], count[MAXCNO];
struct term *original[MAXCNO];
int mult[MAXCNO];
char *alloc();
struct terms msg1;
struct problem msg2;
struct solution msg3;
PROC_ID id;
int msg_type, num_done, length;

new1 = flatten(term1, old_unifier);
new2 = flatten(term2, old_unifier);
elim_dup(new1, new2, old_unifier);

n1 = count_subs(new1);
n2 = count_subs(new2);
if (n1 == 0 && n2 == 0) {
  new_unifiers->first = copy_s(old_unifier);
  new_unifiers->nextu = NULL;
  erase_flat(new1);
  erase_flat(new2);
  return (1);
}
if (n1 == 0 || n2 == 0) {
  erase_flat(new1);
  erase_flat(new2);
  return (0);
}

npairs = 0;
for (ptr = new1; ptr != NULL; ptr = ptr->next) {
  arg = ptr->subterm;
  if (arg != NULL) {
    mult[npairs] = 1;
    original[npairs] = arg;
    for (ptr2 = ptr->next; ptr2 != NULL; ptr2 = ptr2->next) {
      arg2 = ptr2->subterm;
      if (arg2 != NULL && same_term(arg, arg2, old_unifier)) {
        mult[npairs]++;
        ptr2->subterm = NULL;
      }
    }
    npairs++;
  }
}
nA = npairs;

for (ptr = new2; ptr != NULL; ptr = ptr->next) {
```

```
      arg = ptr->subterm;
      if (arg != NULL) {
        mult[npairs] = 1;
        original[npairs] = arg;
        for (ptr2 = ptr->next; ptr2 != NULL; ptr2 = ptr2->next) {
          arg2 = ptr2->subterm;
          if (arg2 != NULL && same_term(arg, arg2, old_unifier)) {
            mult[npairs]++;
            ptr2->subterm = NULL;
          }
        }
        npairs++;
      }
    }
    nB = npairs - nA;

    erase_flat(new1);
    erase_flat(new2);

    num = find_basis(mult, nA, mult+nA, nB, basis);

    if (num < 1 ) exit(1);

    num = weed_basis(num, npairs, basis, original);

    if (num == 0)
      return (0);

    for (i=0; i<npairs; i++)
      count[i] = 0;

    solution = find_solutions(num, npairs, basis, original, 0, so_far, count, NUL
L);

    final = NULL;
    highest = *base;

    if (solution == NULL)
      return(0);

    msg1.ac_op = term1->id;
    msg1.base = *base;
    msg1.nterms = npairs;
    offset = 0;
    for (i = 0; i < npairs; i++)
      offset += trav_term(original[i], &(msg1.list[offset]));
    for(unif = old_unifier; unif != NULL; unif = unif->nexts) {
      msg1.list[offset].type = 'v';
      msg1.list[offset++].id = unif->var;
      offset += trav_term(unif->subst, &(msg1.list[offset]));
    }
    msg1.list[offset++].type = '}';
    length = 4*sizeof(int) + offset*sizeof(struct term_list);
    for (i = 0; i < NSLAVES; i++)
      SEND(&process_ids[i], TERMS, &msg1, length)
```

```
  sol = solution;
  num_done = 0;

  while (num_done < NSLAVES) {
    RECEIVE(&id, &msg_type, &msg3, (match_type(READY) || match_type(SOLUTION)
||
                                  match_type(ERROR_SIGNAL) || match_type(DON
E)))
    if (msg_type == ERROR_SIGNAL) {
      printf("\nError in slave process.  Waiting for all slaves to exit\n");
      stop();
    }
    if (msg_type == SOLUTION) {
      if (msg3.base > highest)
        highest = msg3.base;
      offset = 0;
      for (i = 0; i < msg3.nunifiers; i++) {
        new = (struct unifier *) alloc(sizeof(struct unifier));
        new->nextu = final;
        final = new;
        new->first = NULL;
        while(msg3.list[offset].type != '}') {
          unif = (struct substitution *) alloc(sizeof(struct substitution))
;
          unif->nexts = new->first;
          new->first = unif;
          unif->var = msg3.list[offset++].id;
          unif->subst = parse_t(&(msg3.list[offset]), &j);
          offset += j;
        }
        offset++;
      }
      if (msg3.more)
        SEND(&id, SEND_MORE)
    }
    if ((msg_type == SOLUTION && !msg3.more) || msg_type == READY) {
      if (sol == NULL) {
        SEND(&id, DONE)
      }
      else {
        msg2.nvects = 0;
        for (v = sol->first; v != NULL; v = v->nextv) {
          for (j=0; j<npairs; j++)
            msg2.dsolution[msg2.nvects][j] = basis[v->vect][j+1];
          msg2.nvects++;
        }
        SEND(&id, PROBLEM, &msg2, (1+MAXCNO*msg2.nvects)*sizeof(int))
        sol = sol->next;
      }
    }
    else if (msg_type == DONE)
      num_done++;
  }
  for (sol = solution; sol != NULL; sol = tempsol) {
```

```
    tempsol = sol->next;
    for (v = sol->first; v != NULL; v = tempv) {
      tempv = v->nextv;
      mfree(v, sizeof(*v));
    }
    mfree(sol, sizeof(*sol));
  }
  if (final == NULL)
    return (0);
  new_unifiers->first = final->first;
  new_unifiers->nextu = final->nextu;
  mfree(final,sizeof(*final));
  *base = highest;
  return(1);
}

/******
 *
 * stop kills the slaves and exits. It is used for abnormal exits only.
 *
 ******/
stop()
{
  PROC_ID id;
  int msg_type, i;

  for (i=0; i < NSLAVES; i++)
    SEND(&process_ids[i], END_SIGNAL)
  for (i=0; i < NSLAVES; i++)
    RECEIVE(&id, &msg_type, , match_type(END_SIGNAL))
  WAIT_FOR_END(NSLAVES)
  exit(1);
}
```

The following is the slave mainline including the macro calls for the message-passing primitives. It is called from a driver named *inmain*.

```
#include "types.h"

/******
 *
 *   These macros define the message-passing environment
 *
 ******/

ENV

BEGIN_MSG_TYPES
  MSG_TYPE(TERMS, struct terms {
  int ac_op;
  int base;
  int nterms;
  struct term_list list[TMAXLEN];
}; )
  MSG_TYPE(PROBLEM, struct problem {
  int nvects;
  int dsolution[ZMAXDIMENSION][MAXCNO];
}; )
  MSG_TYPE(SOLUTION, struct solution {
  int base;
  int nunifiers;
  LOGICAL more;
  struct term_list list[6000];
}; )
  MSG_TYPE(SEND_MORE, EMPTY)
  MSG_TYPE(DONE, EMPTY)
  MSG_TYPE(END_SIGNAL, EMPTY)
  MSG_TYPE(READY, EMPTY)
  MSG_TYPE(ERROR_SIGNAL, EMPTY)
END_MSG_TYPES

struct term garbage;

PROC_ID master;

/*****
 *
 *   slave performs the unification on the arguments of the terms being
 *   unified by the master process.  slave is called from inmain.
 *
 *****/
slave()
{
  int msg_type, offset, i, j, k, fail, zbase, length, trav_term(), old_offset;
  PROC_ID id;
  struct terms msg1;
```

```
struct problem msg2;
struct solution msg3;
struct substitution *unif, *u, *temp, *copy_s();
struct term *tempt, *term[MAXCNO], *last, *zterm, *sub, *parse_t();
struct unifier *unifier, *old, *newu, *un, *newu2, *last_un;
char *alloc();
LOGICAL send_flag;
FILE *fopen();

msg_type = READY; /* insure msg_type is not initially END_SIGNAL */
while(msg_type != END_SIGNAL) {
  RECEIVE(&id, &msg_type, &msg1, (match_type(TERMS) || match_type(END_SIGNAL
)))
  COPY_ID(&master, &id)
  if(msg_type == TERMS) {
    offset = 0;
    unif = NULL;
    for (i = 0; i < msg1.nterms; i++) {
      term[i] = parse_t(&(msg1.list[offset]), &j);
      offset += j;
    }
    while(msg1.list[offset].type != '}') {
      u = (struct substitution *) alloc(sizeof(struct substitution));
      u->nexts = unif;
      unif = u;
      u->var = msg1.list[offset++].id;
      u->subst = parse_t(&(msg1.list[offset]), &j);
      offset += j;
    }
    SEND(&id, READY)
    msg_type = PROBLEM;
    while(msg_type == PROBLEM || msg_type == SEND_MORE) {
      RECEIVE(&id, &msg_type, &msg2, (match_type(PROBLEM) || match_type(DO
NE) ||
                                    match_type(END_SIGNAL)))
      if (msg_type == PROBLEM) {
        garbage.child = NULL,
        old = (struct unifier *) alloc(sizeof(struct unifier));
        old->first = copy_s(unif);
        old->nextu = NULL;
        fail = 0;
        zbase = msg1.base + msg2.nvects;
        offset = 0;

        for (i = 0; i < msg1.nterms && !fail; i++) {
          zterm = last = NULL;
          for (j = 0; j < msg2.nvects; j++) {
            for (k = 0; k < msg2.dsolution[j][i]; k++) {
              if (zterm == NULL) {
                zterm = (struct term *) alloc(sizeof(struct term));
                zterm->id = -(msg1.base + j);
                zterm->type = 'v';
                zterm->sibling = garbage.child;
                garbage.child = zterm;
                zterm->child = NULL;
```

```
              }
              else {
                if (last == NULL) {
                  sub = (struct term *) alloc(sizeof(struct term));
                  sub->id = zterm->id;
                  sub->type = 'v';
                  sub->child = sub->sibling = NULL;
                  zterm->child = sub;
                  zterm->id = msg1.ac_op;
                  zterm->type = 't';
                  last = sub;
                }
                sub = (struct term *) alloc(sizeof(struct term));
                last->sibling = sub;
                sub->child = sub->sibling = NULL;
                sub->id = -(msg1.base + j);
                sub->type = 'v';
                last = sub;
              }
            }
          }
          newu = NULL;
          for (unifier = old; unifier != NULL; unifier = unifier->nextu)
{
            newu2 = (struct unifier *) alloc(sizeof(struct unifier));
            if (unify(zterm, term[i], &zbase, unifier->first, newu2)) {
              for (un = newu2; un->nextu != NULL; un = un->nextu);
              un->nextu = newu;
              newu = newu2;
            }
            else
              mfree(newu2, sizeof(struct unifier));
          }
          free_unifiers(old);
          old = newu;
          if (old == NULL)
            fail = 1;
        }
        offset = 0;
        msg3.nunifiers = 0;
        if (fail) {
          msg3.base = msg1.base;
          length = 3*sizeof(int);
        }
        else {
          msg3.base = zbase;
          last_un = NULL;
          send_flag = FALSE;
          for (un = old; un != NULL && msg_type != END_SIGNAL; un = un->
nextu) {
            old_offset = offset;
            for (u = un->first; u != NULL; u = u->nexts) {
              msg3.list[offset].type = 'v';
              msg3.list[offset++].id = u->var;
              offset += trav_term(u->subst, &(msg3.list[offset]));
```

```
                }
                msg3.list[offset++].type = '}';
                msg3.nunifiers++;
                length = 3*sizeof(int) + offset*sizeof(struct term_list);
                if (length > 5000) {
                  if (last_un == NULL)
                    stop();
                  un = last_un;
                  msg3.nunifiers--;
                  offset = old_offset;
                  send_flag = TRUE;
                  length = 3*sizeof(int) + offset*sizeof(struct term_list)

                }
                last_un = un;
                if (send_flag || ((length > (5000 - UMAXLEN)) &&
                               (un->nextu != NULL))) {
                  msg3.more = TRUE;
                  SEND(&id, SOLUTION, &msg3, length)
                  RECEIVE(&id, &msg_type, ,(match_type(SEND_MORE) ||
                                    match_type(END_SIGNAL)))
                  offset = msg3.nunifiers = 0;
                  last_un = NULL;
                  send_flag = FALSE;
                }
              }
              free_unifiers(old);
            }
            msg3.more = FALSE;
            SEND(&id, SOLUTION, &msg3, length)

            for (sub = garbage.child; sub != NULL; sub = tempt) {
              tempt = sub->sibling;
              erase_term(sub);
            }
          }
          else if (msg_type == DONE)
            SEND(&id, DONE)
        }
        for (u = unif; u != NULL; u = temp) {
          temp = u->nexts;
          erase_term(u->subst);
          mfree(u, sizeof(struct substitution));
        }
        for (i = 0; i < msg1.nterms; i++)
          erase_term(term[i]);
      }
    }
    SEND(&id, END_SIGNAL)
}

/******
 *
 *   stop provides a graceful abort.  It signals to the master that an error
 *   has occurred and terminates normally.
```

```
 *
 ******/
stop()
{
  PROC_ID id;
  int msg_type;

  SEND(&master, ERROR_SIGNAL)
  RECEIVE(&id, &msg_type, , match_type(END_SIGNAL))
  SEND(&id, END_SIGNAL)

  /*  copied from program "inmain" since we are not returning from "slave"  */
  xx_kill_daemon();
  xx_kill_grandkids();
  xx_locks_dealloc();
  xx_mem(3);

  exit(1);
}
```

The following is the sequential version of the AC unification algorithm which is used by the slave processes if AC functions are arguments to the function being unified in parallel.

```
#include "types.h"

extern struct term garbage;

/********
 *
 * AC_unify is the sequential version of Stickel's algorithm. It is used by
 * the slave processes when AC functions are nested.
 *
 ********/
int AC_unify(term1, term2, base, old_unifier, new_unifiers)
struct term *term1, *term2;
int *base;
struct substitution *old_unifier;
struct unifier *new_unifiers;
{
   void elim_dup(), erase_term();
   struct flat *new1, *new2, *flatten(), *ptr2, *ptr;
   struct term *sub, *zterm;
   struct term *arg, *arg2, *last;
   struct substitution *copy_s();
   struct vector *v, *tempv;
   struct unifier *newu, *newu2, *final, *u, *old, *unif;
   int n1, n2, nA, nB, same_term(), count_subs();
   int find_basis(), i, j, fail, number;
   int npairs, num, basis[ZMAXDIMENSION][MAXCNO+1];
   int weed_basis(), zbase, highest;
   struct dio_soln *solution, *sol, *find_solutions(), *tempsol;
   int so_far[ZMAXDIMENSION], count[MAXCNO];
   struct term *original[MAXCNO];
   int mult[MAXCNO];
   char *alloc();

   new1 = flatten(term1, old_unifier);
   new2 = flatten(term2, old_unifier);
   elim_dup(new1, new2, old_unifier);

   n1 = count_subs(new1);
   n2 = count_subs(new2);
   if(n1 == 0 && n2 == 0) {
      new_unifiers->first = copy_s(old_unifier);
      new_unifiers->nextu = NULL;
      erase_flat(new1);
      erase_flat(new2);
      return (1);
   }
   if(n1 == 0 || n2 == 0) {
```

```
    erase_flat(new1);
    erase_flat(new2);
    return (0);
}

npairs = 0;
for (ptr = new1; ptr != NULL; ptr = ptr->next) {
    arg = ptr->subterm;
    if (arg != NULL) {
        mult[npairs] = 1;
        original[npairs] = arg;
        for (ptr2 = ptr->next; ptr2 != NULL; ptr2 = ptr2->next) {
            arg2 = ptr2->subterm;
            if (arg2 != NULL && same_term(arg, arg2, old_unifier)) {
                mult[npairs]++;
                ptr2->subterm = NULL;
            }
        }
        npairs++;
    }
}
nA = npairs;

for (ptr = new2; ptr != NULL; ptr = ptr->next) {
    arg = ptr->subterm;
    if (arg != NULL) {
        mult[npairs] = 1;
        original[npairs] = arg;
        for (ptr2 = ptr->next; ptr2 != NULL; ptr2 = ptr2->next) {
            arg2 = ptr2->subterm;
            if (arg2 != NULL && same_term(arg, arg2, old_unifier)) {
                mult[npairs]++;
                ptr2->subterm = NULL;
            }
        }
        npairs++;
    }
}
nB = npairs - nA;

erase_flat(new1);
erase_flat(new2);

num = find_basis(mult, nA, mult+nA, nB, basis);

if (num < 1 ) exit(1);

num = weed_basis(num, npairs, basis, original);

if (num == 0)
    return (0);

for (i=0; i<npairs; i++)
    count[i] = 0;
```

```
  solution = find_solutions(num, npairs, basis, original, 0, so_far, count, NUL
L);

  final = NULL;
  highest = *base;

  for (sol = solution; sol != NULL; sol = sol->next) {
    old = (struct unifier *) alloc(sizeof(struct unifier));
    old->first = copy_s(old_unifier);
    old->nextu = NULL;
    fail = 0;
    zbase = *base + num;

    for (i=0; i<npairs && !fail; i++) {
      zterm = last = NULL;
      for (v = sol->first; v!=NULL; v = v->nextv) {
        number = v->vect;
        if (basis[number][i+1] > 0)
          for (j = 0; j< basis[number][i+1]; j++) {
            if (zterm == NULL) {
              zterm = (struct term *) alloc(sizeof(struct term));
              zterm->id = -(*base+number);
              zterm->type = 'v';
              zterm->sibling = garbage.child;
              garbage.child = zterm;
              zterm->child = NULL;
            }
            else {
              if (last == NULL) {
                sub = (struct term *) alloc(sizeof(struct term));
                sub->id = zterm->id;
                sub->type = 'v';
                sub->child = sub->sibling = NULL;
                zterm->child = sub;
                zterm->id = term1->id;
                zterm->type = 't';
                last = sub;
              }
              sub = (struct term *) alloc(sizeof(struct term));
              last->sibling = sub;
              sub->child = sub->sibling = NULL;
              sub->id = -(*base+number);
              sub->type = 'v';
              last = sub;
            }
          }
      }
      newu = NULL;
      for (unif = old; unif != NULL; unif=unif->nextu) {
        newu2 = (struct unifier *) alloc(sizeof(struct unifier));
        if (unify(zterm, original[i], &zbase, unif->first, newu2)) {
          for(u = newu2; u->nextu != NULL; u = u->nextu);
          u->nextu = newu;
          newu = newu2;
        }
```

```
            else
               mfree(newu2, sizeof(*newu2));
         }
         free_unifiers(old);
         old = newu;
         if (old == NULL)
            fail = 1;
      }
      if (!fail) {
         for (u = old; u->nextu != NULL; u = u->nextu);
         u->nextu = final;
         final = old;
         if (zbase > highest)
            highest = zbase;
      }
   }
   for (sol = solution; sol != NULL; sol = tempsol) {
      tempsol = sol->next;
      for (v = sol->first; v != NULL; v = tempv) {
         tempv = v->nextv;
         mfree(v, sizeof(*v));
      }
      mfree(sol, sizeof(*sol));
   }
   if (final == NULL)
      return (0);
   new_unifiers->first = final->first;
   new_unifiers->nextu = final->nextu;
   mfree(final,sizeof(*final));
   *base = highest;
   return(1);
}
```

The following code is common to both the master and slave processes.

```
#include "types.h"

/*****
 *
 *   parse_term reads symbols from the standard input and builds a term.
 *
 *****/
int parse_term(tree, symtab, nsymbols)
struct term *tree;
char symtab[][10];
int *nsymbols;
{
  struct term *subtree, *last;
  char string[10], *alloc();
  int i, done, found;

  scanf("%s",string);
  if (string[0] == EOF) {
    printf("\n\nInvalid syntax!\n\n");
    return (1);
  }
  found = 0;
  for (i=0; i<*nsymbols && !found; i++)
    if (str_equal(string, symtab[i])) {
      tree->id = i;
      found = 1;
    }
  if (!found) {
    for (i=0; (symtab[*nsymbols][i] = string[i]) != '\0'; i++);
    tree->id = (*nsymbols)++;
  }
  tree->type = 't';

  last = NULL;
  done = 0;

  while (!done) {
    scanf("%s",string);
    if (string[0] == EOF) {
      printf("\n\nInvalid syntax!\n\n");
      return(1);
    }

    if (string[0] == ')') {
      if (last == NULL)
        printf("\nInvalid syntax");
      else
        last->sibling = NULL;
      done = 1;
    }
    else {
```

```
        subtree = (struct term *) alloc(sizeof(struct term));
        if (last == NULL)
          tree->child = subtree;
        else
          last->sibling = subtree;

        if (string[0] == '(') {
          if (parse_term(subtree, symtab, nsymbols))
            return(1);
        }
        else {
          found = 0;
          for (i=0; i<*nsymbols; i++)
            if (str_equal(string, symtab[i])) {
              subtree->id = i;
              found = 1;
            }
          if (!found) {
            for (i=0; (symtab[*nsymbols][i] = string[i]) != '\0'; i++);
            subtree->id = (*nsymbols)++;
          }
          if (string[0] >= 'u' && string[0] <= 'z')
            subtree->type = 'v';
          else
            subtree->type = 'c';
          subtree->child = NULL;
        }
        last = subtree ;
      }
    }
    return(0);
}

/*****
 *
 *  print_term prints a term to standard output
 *
 *****/
print_term(tree, unif, symtab)
struct term *tree;
struct substitution *unif;
char symtab[][10];
{
  struct term *subtree, *sub, *sub_v();
  char *build_z(), *temp;

  if (tree->type == 'v')
    tree = sub_v(tree, unif);

  if (tree->child == NULL)
    printf("%s",symtab[tree->id]);
  else {
    printf("(%s",symtab[tree->id]);
    subtree = tree->child;
```

```
      while (subtree != NULL) {
        printf(" ");
        if (subtree->type == 'v')
          sub = sub_v(subtree, unif);
        else
          sub = subtree;

        if (sub->child == NULL)
          if (sub->id < 0) {
            printf("%s",(temp = build_z(-sub->id)));
            mfree(temp, 10*sizeof(char));
          }
          else
            printf("%s",symtab[sub->id]);
        else
          print_term(sub, unif, symtab);
        subtree = subtree->sibling;
      }
      printf(")");
    }
    return;
}

/*****
 *
 *  print_unifiers prints up to 100 unifiers to standart output and returns
 *  the total number of unifiers
 *
 *****/
int print_unifiers(unifiers, symtab)
struct unifier *unifiers;
char symtab[][10];
{
    struct unifier *temp;
    struct substitution *subs;
    int count = 0;

    for (temp = unifiers; temp != NULL; temp = temp->nextu) {
      if (count++ < 100) {
        printf ("\n{ ");
        for (subs = temp->first; subs != NULL; subs = subs->nexts)
          if ((subs->var)>=0) {
            printf("%s/", symtab[subs->var]);
            print_term(subs->subst, temp->first, symtab);
            printf(", ");
          }
        printf("}");
      }
    }
    return(count);
}

/*****
 *
 *  sub_v does substitutions on a variable if the variable is in the domain
```

```
 *   of a unifier.  It returns a pointer to the final term.
 *
 *****/
struct term *sub_v(node, unif)
struct term *node;
struct substitution *unif;
{
  struct substitution *v;
  struct term *t;

  for (v = unif; v != NULL; v = v->nexts)
    if (v->var == node->id) {
      t = v->subst;
      if (t->type == 'v')
        return (sub_v(t, unif));
      else
        return (t);
    }
  return (node);
}
/*****
 *
 *   flatten flattens a term and returns a pointer to the flattened term
 *
 *****
struct flat *flatten(tree, unif)
struct term *tree;
struct substitution *unif;
{
  struct term *sub_v(), *subtree, *ptr;
  struct flat *temp, *last, *root;
  char *alloc();

  if (tree->type == 'v')
    tree = sub_v(tree, unif);

  last = NULL;
  for (ptr = tree->child; ptr != NULL; ptr = ptr->sibling) {
    if (ptr->type == 'v')
      subtree = sub_v(ptr, unif);
    else
      subtree = ptr;
    if (subtree->id == tree->id) {
      temp = flatten(subtree, unif);
      if (last == NULL)
        root = temp;
      else
        last->next = temp;
      for (last = temp; last->next != NULL; last = last->next);
    }
    else {
      temp = (struct flat *) alloc(sizeof(struct flat));
      temp->subterm = subtree;
      temp->next = NULL;
      if (last == NULL)
```

```
        root = temp;
      else
        last->next = temp;
      last = temp;
    }
  }
  return (root);
}

/*****
 *
 *  str_equal tests if two character strings are equal
 *
 *****/
int str_equal(s1, s2)
char s1[], s2[];
{
  int i;

  for (i = 0; s1[i] == s2[i]; i++)
    if (s1[i] == '\0')
      return(1);

  return(0);
}

/*****
 *
 *  occurs performs the occurs check.  It returns true if var occurs in the
 *  term tree
 *
 *****/
int occurs(var, tree, unif, checked, nchecked)
int var, checked[], *nchecked;
struct term *tree;
struct substitution *unif;
{
  struct term *subtree;
  int i;
  struct substitution *v;

  if (tree->type == 'v')
    if (tree->id == var) {
      return (1);
    }
    else {
      for (i = 0; i < *nchecked; i++)
        if (tree->id == checked[i])
          return (0);

      checked[(*nchecked)++] = tree->id;

      for (v = unif; v != NULL; v = v->nexts)
        if (tree->id == v->var)
          return (occurs(var, v->subst, unif, checked, nchecked));
```

```
        return (0);
      }
    else
      for (subtree = tree->child; subtree != NULL; subtree = subtree->sibling)
        if (occurs(var, subtree, unif, checked, nchecked))
          return (1);
    return(0);
}

/*****
 *
 *  unify determines which unification algorithm is used to unify two terms.
 *
 *****/
int unify(term1, term2, base, old_unifier, new_unifiers)
struct term *term1, *term2;
int *base;
struct substitution *old_unifier;
struct unifier *new_unifiers;
{
    struct term *sub_v();
    int R_unify(), AC_unify();
    int is_AC();

    if (term1->type == 'v')
      term1 = sub_v(term1, old_unifier);
    if (term2->type == 'v')
      term2 = sub_v(term2, old_unifier);

    if (term1->child == NULL || term2->child == NULL)
      return (R_unify(term1, term2, base, old_unifier, new_unifiers));
    else if (term1->id == term2->id)
      if (is_AC(term1->id))
        return (AC_unify(term1, term2, base, old_unifier, new_unifiers));
      else
        return (R_unify(term1, term2, base, old_unifier, new_unifiers));
    else
      return (0);
}

/*****
 *
 *  is_AC returns true if the function func is AC
 *
 *****/
int is_AC(func)
int func;
{
    return (func < NUMAC);
}

/****
 *
 *  R_unify is Robinson's unification algorithm
 *
```

```
 *****/
int R_unify(term1, term2, base, old_unifier, new_unifiers)
struct term *term1, *term2;
int *base;
struct substitution *old_unifier;
struct unifier *new_unifiers;
{
  void free_unifiers();
  int checked[100], nchecked;
  int count_subterms(), unify();
  struct unifier *old, *new, *new2, *temp, *unif;
  struct substitution *new_unif,*copy_s();
  struct term *sub1, *sub2;
  char *alloc();

  if (term1->type == 'v') {
    if (term2->type == 'v' && term1->id == term2->id) {
      new_unifiers->first = copy_s(old_unifier);
      new_unifiers->nextu = NULL;
      return (1);
    }
    nchecked = 0;
    if (occurs(term1->id, term2, old_unifier, checked, &nchecked))
      return(0);
    new_unif = (struct substitution *) alloc(sizeof(struct substitution));
    new_unif->var = term1->id;
    new_unif->subst = term2;
    new_unif->nexts = copy_s(old_unifier);
    new_unifiers->first = new_unif;
    new_unifiers->nextu = NULL;
    return(1);
  }
  else if (term2->type == 'v')
    return (R_unify(term2, term1, base, old_unifier, new_unifiers));
  else if (term1->id == term2->id) {
    if (count_subterms(term1) == count_subterms(term2)) {
      old = (struct unifier *) alloc(sizeof(struct unifier));
      old->first = copy_s(old_unifier);
      old->nextu = NULL;
      sub2 = term2->child;
      for (sub1 = term1->child; sub1 != NULL; sub1 = sub1->sibling) {
        new = NULL;
        for (unif = old; unif != NULL; unif = unif->nextu) {
          new2 = (struct unifier *) alloc(sizeof(struct unifier));
          if (unify(sub1, sub2, base, unif->first, new2)) {
            for (temp=new2; temp->nextu != NULL; temp = temp->nextu);
            temp->nextu = new;
            new = new2;
          }
          else
            mfree(new2, sizeof(struct unifier));
        }
        free_unifiers(old);
        old = new;
        sub2 = sub2->sibling;
```

```
          if (old == NULL)
            return(0);
        }
        new_unifiers->first = old->first;
        new_unifiers->nextu = old->nextu;
        mfree (old, sizeof(*old));
        return(1);
      }
    }
    return (0);
}

/*****
 *
 *  copy_s makes a copy of a substitution
 *
 *****/
struct substitution *copy_s(old)
struct substitution *old;
{
    struct substitution *new;
    char *alloc();

    if (old == NULL)
      new = NULL;
    else {
      new = (struct substitution *) alloc(sizeof(struct substitution));
      new->var = old->var;
      new->subst = old->subst;
      new->nexts = copy_s(old->nexts);
    }
    return (new);
}

/*****
 *
 *  count_subs returns the number of subterms in a flattened term
 *
 *****/
int count_subs(f)
struct flat *f;
{
    int i;
    struct flat *temp;

    i = 0;
    for (temp = f; temp != NULL; temp = temp->next)
      if (temp->subterm != NULL)
        i++;
    return(i);
}

/*****
 *
 * count_subterms returns the number of argument in a term
```

```
 *
 *****/
int count_subterms(t)
struct term *t;
{
  int i;
  struct term *temp;

  i = 0;
  for (temp = t->child; temp != NULL; temp = temp->sibling)
    i++;
  return(i);
}

/*****
 *
 *  free_unifiers deallocated the memory used by u
 *
 *****/
void free_unifiers(u)
struct unifier *u;
{
  struct unifier *temp, *next;
  struct substitution *sub, *sub2;

  for (temp = u; temp != NULL; temp = next) {
    for (sub = temp->first; sub != NULL; sub = sub2) {
      sub2 = sub->nexts;
      mfree(sub, sizeof(*sub));
    }
    next = temp->nextu;
    mfree(temp, sizeof(*temp));
  }
}

/*****
 *
 *  same_term returns true if term1 and term2 are identical
 *
 *****
int same_term(term1, term2, unif)
struct term *term1, *term2;
struct substitution *unif;
{
  int count_subterms();
  struct term *sub1, *sub2, *sub_v();

  if (term1->type == 'v') {
    if (term2->type == 'v' && term1->id == term2->id)
      return(1);
    term1 = sub_v(term1, unif);
  }

  if (term2->type == 'v')
    term2 = sub_v(term2, unif);
```

```
  if (term1->id == term2->id)
    if (count_subterms(term1) == count_subterms(term2)) {
      for (sub1 = term1->child, sub2 = term2->child; sub1 != NULL;
           sub1 = sub1->sibling, sub2 = sub2->sibling)
         if (!same_term(sub1, sub2, unif))
           return (0);
      return (1);
    }
  return (0);
}

/*****
 *
 * erase_flat deallocates the memory used by a flattened term
 *
 *****/
erase_flat(f)
struct flat *f;
{
  struct flat *temp, *ff;

  for (ff = f; ff != NULL; ff = temp) {
    temp = ff->next;
    mfree(ff, sizeof(*ff));
  }
}

/*****
 *
 * erase_term deallocates the memory used by a term
 *
 *****/
void erase_term(t)
struct term *t;
{
  struct term *subterm, *temp;

  if (t != NULL) {
    for (subterm = t->child; subterm != NULL; subterm = temp) {
      temp = subterm->sibling;
      erase_term(subterm);
    }
    mfree(t, sizeof(*t));
  }
}

/*****
 *
 *  elim_dup removes similar terms from both terms
 *
 *****/
void elim_dup(term1, term2, unif)
struct flat *term1, *term2;
```

```
struct substitution *unif;
{
    struct term *sub1, *sub2;
    struct flat *ptr1, *ptr2;
    int done, same_term();

    for (ptr1 = term1; ptr1 != NULL; ptr1 = ptr1->next) {
        sub1 = ptr1->subterm;
        if (sub1 != NULL) {
            done = 0;
            for (ptr2 = term2; ptr2 != NULL && !done; ptr2 = ptr2->next) {
                sub2 = ptr2->subterm;
                if (sub2 != NULL && same_term(sub1, sub2, unif)) {
                    ptr1->subterm = NULL;
                    ptr2->subterm = NULL;
                    done = 1;
                }
            }
        }
    }
}

/*****
 *
 *  build_z creates a character string containing the name of an introduced
 *  variable
 *
 *****/
char *build_z(number)
int number;
{
    int i, index, num2;
    char temp[10], *id, *alloc();

    id = alloc(sizeof(char)*10);
    id[0] = 'z';
    index = 0;
    while (number != 0) {
        num2 = number/10;
        temp[index++] = '0' + number - 10*num2;
        number = num2;
    }
    for (i = 0; i< index; i++)
        id[i+1] = temp[index-i-1];
    id[index+2] = '\0';

    return (id);
}

/*****
 *
 *  print_basis is a debugging tool that prints the basis of solutions to a
 *  Diophantine equation
 *
```

```
 *****/
print_basis(basis, num, npairs)
int basis[][MAXCNO+1], num, npairs;
{
  int i, j;

  printf("\n\nBasis:\n\n");
  for (i=0; i<num; i++) {
    for (j=1; j<=npairs; j++)
      printf("%d ",basis[i][j]);
    printf("\n");
  }
  printf("\n");
}

/*****
 *
 * weed_basis removes basis solutions which can not appear in any valid
 * solution to the Diophantine equation
 *
 *****/

int weed_basis(num, npairs, basis, original)
int num, npairs, basis[][MAXCNO+1];
struct term *original[];
{
  int i, j, newnum, good, cflag, tflag, termid;

  newnum = 0;
  for (i=0; i<num; i++) {
    good = 1;
    cflag = 0;
    tflag = 0;
    for (j=0; j<npairs && good; j++) {
      if (original[j]->type == 'c') {
        if(basis[i][j+1] > 1)
          good = 0;
        else if(basis[i][j+1] == 1) {
          if (cflag || tflag)
            good = 0;
          else
            cflag = 1;
        }
      }
      else if (original[j]->type == 't') {
        if(basis[i][j+1] > 1)
          good = 0;
        else if(basis[i][j+1] == 1) {
          if (cflag || (tflag && original[j]->id != termid))
            good = 0;
          else {
            tflag = 1;
            termid = original[j]->id;
          }
        }
```

```
            }
        }
        if (good) {
            if (i != newnum)
                for (j=0; j<npairs; j++)
                    basis[newnum][j+1] = basis[i][j+1];
            newnum++;
        }
    }
    return (newnum);
}

/*****
 *
 *  find_solutions generates the valid solutions to the Diophantine
 *  equation, given the weeded basis
 *
 *****/
struct dio_soln *find_solutions(num, npairs, basis, original, nb, so_far, count,
 solution)
int num, npairs, basis[][MAXCNO+1], nb, so_far[], count[MAXCNO];
struct dio_soln *solution;
struct term *original[];
{
    int i, j, illegal, good;
    struct vector *v;
    struct dio_soln *new_solution;
    char *alloc();

    num--;
    for (i=0; i<2; i++) {
        illegal = 0;
        if (i == 1) {
            so_far[nb++] = num;
            for (j = 0; j<npairs; j++) {
                count[j] += basis[num][j+1];
                if (count[j] > 1 && original[j]->type != 'v')
                    illegal = 1;
            }
        }
        if (!illegal) {
            if (num == 0) {
                good = 1;
                for (j=0; j<npairs; j++)
                    if (count[j] == 0)
                        good = 0;
                if (good) {
                    new_solution = (struct dio_soln *) alloc (sizeof(struct dio_soln)
);
                    new_solution->next = solution;
                    new_solution->first = NULL;
                    solution = new_solution;
                    for (j=0; j<nb; j++) {
                        v = (struct vector *) alloc(sizeof(struct vector));
                        v->nextv = solution->first;
```

```
                solution->first = v;
                v->vect = so_far[j];
            }
          }
        }
        else
          solution = find_solutions(num, npairs, basis, original, nb, so_far,
                                    count, solution);
      }
    }
    for (i = 0; i<npairs; i++)
      count[i] -= basis[num][i+1];

    return(solution);
}

struct set {
    struct set *next;
    unsigned char num[10];
} *C[10][10];

/*****
 *
 *  find_basis returns the basis solutions of a Diophantine equation
 *
 *****/
int find_basis(A, nA, B, nB, basis)
int A[MAXCNO], B[MAXCNO], basis[ZMAXDIMENSION][MAXCNO+1], nA, nB;
{
    /*  Zhang's algorithm  (simple case )  */

    LOGICAL A_is_all_ones, B_is_all_ones;
    struct set *find_C(), *ptr;
    int i, j, nvects, num_ones, num_other, ones_index, other_index, *other;

    A_is_all_ones = TRUE;

    for (i = 0; i < nA && A_is_all_ones; i++)
        if (A[i] != 1)
            A_is_all_ones = FALSE;

    if (A_is_all_ones) {
        num_ones = nA;
        ones_index = 1;
        num_other = nB;
        other = B;
        other_index = 1+nA;
    }
    else {
        B_is_all_ones = TRUE;

        for (i = 0; i < nB && B_is_all_ones; i++)
            if (B[i] != 1)
                B_is_all_ones = FALSE;
```

```
        if (B_is_all_ones) {
            num_ones = nB;
            ones_index = 1+nA;
            num_other = nA;
            other = A;
            other_index = 1;
        }
        else
            return(lankford(A, nA, B, nB, basis));
    }

    nvects = 0;
    for (i = 0; i < num_other; i++)
        for (ptr = find_C(num_ones, other[i]); ptr != NULL; ptr = ptr->next) {
            for (j = 0; j < num_ones; j++)
                basis[nvects][ones_index+j] = ptr->num[j];
            for (j = 0; j < num_other; j++)
                basis[nvects][other_index+j] = ( i == j);
            nvects++;
        }

    return(nvects);
}

/*****
 *
 *  find_C returns the set C(m, k) used in Zhang's algorithm
 *
 *****/
struct set *find_C(m, k)
int m, k;
{
    char *alloc();
    int i, j;
    struct set *ptr, *ptr2;

    if (m > 9 || k > 9) {
        printf("\n argument to find_C is too big !\n\n");
        return(NULL);
    }
    if (C[m][k] == NULL) {
        if (m == 1) {
            C[1][k] = (struct set *) alloc(sizeof(char *) + sizeof(char));
            C[1][k]->next = NULL;
            C[1][k]->num[0] = k;
        }
        else {
            for (i = 0; i <= k; i++)
                for (ptr = find_C(m-1, k-i); ptr != NULL; ptr = ptr->next) {
                    ptr2 = (struct set *) alloc(sizeof(char *) + m*sizeof(char))
;
                    ptr2->next = C[m][k];
                    C[m][k] = ptr2;
                    for (j = 0; j < m-1; j++)
                        ptr2->num[j] = ptr->num[j];
```

```
                ptr2->num[m-1] = i;
            }
        }
    }
    return(C[m][k]);
}

/*****
 *
 *  init_C initializes the array used to calculate C
 *
 *****
init_C()
{
    int i, j;

    for (i = 0; i < 10; i++)
        for (j = 0 ; j < 10; j++)
            C[i][j] = NULL;
}

/*****
 *
 *  lankford finds the basis solutions of a Diophantine equation using
 *  Lankford's algorithm
 *
 *****/
int lankford(A, nA, B, nB, Zmatrix)
int A[MAXCNO], B[MAXCNO], Zmatrix[ZMAXDIMENSION][MAXCNO+1], nA, nB;
{
    int Nmatrix[NMAXDIMENSION][MAXCNO+1], Pmatrix[NMAXDIMENSION][MAXCNO+1]
    int new[MAXCNO+1];
    int nN, nP, nZ, n, old_offset, new_offset;
    int i, j, jj, k;
    int Nend, Pend, Zend;
    int newN(), newZ(), newP();

    n = nA + nB + 1;
    nN = nZ = nP = 0;
    for (i=1; i<n; i++)
        new[i] = 0;
    for (i=0; i<nA; i++) {
        new[i+1] = 1;
        for (j=0; j < nB; j++) {
            jj = nA + 1 + j;
            new[jj] = 1;
            new[0] = A[i] - B[j];
            if (new[0] < 0) {
                for (k=0; k<n; k++)
                    Nmatrix[nN][k] = new[k];
                nN++;
            }
            else if (new[0] == 0) {
                for (k=0; k<n; k++)
                    Zmatrix[nZ][k] = new[k];
```

```
            nZ++;
        }
        else {
            for (k=0; k<n; k++)
                Pmatrix[nP][k] = new[k];
            nP++;
        }
        new[jj] = 0;
    }
    new[i+1] = 0;
}

new_offset = 0;
old_offset = NMAXDIMENSION/2;

while (nN + nP > 0) {
    i = old_offset;
    old_offset = new_offset;
    new_offset = i;
    Nend = nN;
    Pend = nP;
    Zend = nZ;
    nN = nP = 0;
    for (i = 0; i < Nend; i++) {
        for (j = 1; j < n; j++)
            new[j] = Nmatrix[i+old_offset][j];
        for (j = 0; j < nA; j++) {
            new[j+1]++;
            new[0] = A[j] + Nmatrix[i+old_offset][0];
            if (new[0] < 0) {
                nN = newN(Nmatrix, nN, Zmatrix, nZ, new, n, new_offset);
                if (nN == -1)
                    return(0);
            }
            else if(new[0] == 0) {
                if ((nZ = newZ(Zmatrix, nZ, Zend, new, n)) == -1)
                    return(0);
            }
            else {
                nP = newP(Pmatrix, nP, Zmatrix, nZ, new, n, new_offset);
                if (nP == -1)
                    return(0);
            }
            new[j+1]--;
        }
    }
    for (i = 0; i < Pend; i++) {
        for (j = 1; j < n; j++)
            new[j] = Pmatrix[old_offset+i][j];
        for (j = 0; j < nB; j++) {
            new[j+1+nA]++;
            new[0] = Pmatrix[old_offset+i][0] - B[j];
            if (new[0] < 0) {
                nN = newN(Nmatrix, nN, Zmatrix, nZ, new, n, new_offset);
                if (nN == -1)
```

```
            return(0);
          }
          else if(new[0] == 0) {
            if ((nZ = newZ(Zmatrix, nZ, Zend, new, n)) == -1)
              return(0);
          }
          else {
            nP = newP(Pmatrix, nP, Zmatrix, nZ, new, n, new_offset);
            if (nP == -1)
              return(0);
          }
          new[j+1+nA]--;
        }
      }
    }
    return(nZ);
}

/******
 *
 * newN adds a row the Lankfords N-matrix
 *
 ******/
int newN(Nmatrix, nN, Zmatrix, nZ, new, n, offset)
int Nmatrix[][MAXCNO+1], nN, offset;
int Zmatrix[][MAXCNO+1], nZ, new[MAXCNO+1], n;
{
    int i, j, reducible, same;

    same = 0;
    for (i=0; i < nN && !same; i++) {
      same = 1;
      for (j=1; j < n && same; j++)
        if (new[j] != Nmatrix[offset+i][j])
          same = 0;
    }
    if (!same) {
      reducible = 0;
      for (i=0; i < nZ && !reducible; i++) {
        reducible = 1;
        for (j=1; j < n && reducible; j++)
          if (new[j] < Zmatrix[i][j])
            reducible = 0;
      }
    }
    if (!same && !reducible) {
      if (nN == NMAXDIMENSION/2)
        nN = -1;
      else {
        for (i=0; i<n; i++)
          Nmatrix[offset+nN][i] = new[i];
        nN++;
      }
    }
```

```
  return(nN);
}

/*****
 *
 *   newZ adds a row to Lankford's Z-matrix which becomes the basis
 *
 *****/
int newZ(Zmatrix, nZ, Zend, new, n)
int Zmatrix[][MAXCNO+1], nZ, Zend, new[MAXCNO+1], n;
{
  int i, j, same;

  same = 0;
  for (i=Zend; i < nZ && !same; i++) {
    same = 1;
    for (j=1; j < n && same; j++)
      if (new[j] != Zmatrix[i][j])
        same = 0;
  }
  if (!same) {
    if (nZ == ZMAXDIMENSION)
      nZ = -1;
    else {
      for (i=0; i<n; i++)
        Zmatrix[nZ][i] = new[i];
      nZ++;
    }
  }

  return(nZ);
}

/*****
 *
 *   newP adds a row to Lankford's P-matrix
 *
 *****/
int newP(Pmatrix, nP, Zmatrix, nZ, new, n, offset)
int Pmatrix[][MAXCNO+1], nP, offset;
int Zmatrix[][MAXCNO+1], nZ, new[MAXCNO+1], n;
{
  int i, j, reducible, same;

  same = 0;
  for (i=0; i < nP && !same; i++) {
    same = 1;
    for (j=1; j < n && same; j++)
      if (new[j] != Pmatrix[offset+i][j])
        same = 0;
  }
  if (!same) {
    reducible = 0;
    for (i=0; i < nZ && !reducible; i++) {
      reducible = 1;
```

```
        for (j=1; j < n && reducible; j++)
          if (new[j] < Zmatrix[i][j])
            reducible = 0;
      }
    }
    if (!same && !reducible) {
      if (nP == NMAXDIMENSION/2)
        nP = -1;
      else {
        for (i=0; i<n; i++)
          Pmatrix[offset+nP][i] = new[i];
        nP++;
      }
    }

    return(nP);
}

#define FACTOR 1000
#define FACT4 4*FACTOR

struct root {
    int allocated;
    char *base;
    char **empty;
} class[10];

/*****
 *
 *   alloc dynamically allocates and manages memory
 *
 *****/
char *alloc(size)
unsigned size;
{
    char *malloc();
    char *ptr, **ptr2;
    int index;

    /* blocks are in multiples of 4 bytes */
    index = (size-1)>>2;

    if (index > 9)
      stop();

    if (class[index].base == NULL) {
      class[index].allocated = 0;
      class[index].base = malloc(FACT4*(index+1));
      class[index].empty = NULL;
    }

    if (class[index].empty == NULL) {
      if (class[index].allocated == FACTOR) {
        class[index].allocated = 1;
        ptr = class[index].base = malloc(FACT4*(index+1));
```

```
            if (ptr == NULL) {
                printf("\nMALLOC returned NULL!!!\n");
                stop();
            }
        }
        else
            ptr = class[index].base + (((index+1) * class[index].allocated++)<<2);
    }
    else {
        ptr2 = class[index].empty;
            class[index].empty = (char **) *ptr2;
        ptr = (char *) ptr2;
    }
    return (ptr);
}

/*****
 *
 *  mfree keeps track of memory no longer needed for future use by alloc
 *
 *****/
mfree(block, size)
char **block;
unsigned size;
{
    int index;

    /* blocks are in multiples of 4 bytes */
    index = (size-1)>>2;

    *block = (char *) class[index].empty;
    class[index].empty = block;
}

/*****
 *
 *  trav_term traverses a term and copies it into a list
 *
 *****/
int trav_term(term, list)
struct term *term;
struct term_list list[];
{

    struct term *subterm;
    int i;

    list[0].type = term->type;
    list[0].id = term->id;
    i = 1;

    if (term->type == 't') {
        for (subterm = term->child; subterm != NULL; subterm = subterm->sibling)
            i += trav_term(subterm, &list[i]);
        list[i++].type = ')';
```

```
    }

    return(i);
}

/*****
 *
 *   parse_t parses a term from a list
 *
 *****/
struct term *parse_t(list, next)
struct term_list list[];
int *next;
{

    struct term *term, *subterm, *last;
    int i;
    char *alloc();

    term = (struct term *) alloc(sizeof(struct term));
    term->type = list[0].type;
    term->id = list[0].id;
    term->child = NULL;
    term->sibling = NULL;
    last = NULL;
    *next = 1;

    if (list[0].type == 't') {
        while (list[*next].type != ')') {
            subterm = parse_t(&list[*next], &i);
            (*next) += i;
            if (last == NULL)
                term->child = subterm;
            else
                last->sibling = subterm;
            last = subterm;
        }
        (*next)++;
    }
    return (term);
}
```

The clock routines used to time the programs follow.

```
#include "types.h"

struct clock clocks[MAX_CLOCKS];

/*************
 *
 *    clock_init() - Initialize all clocks.
 *
 *************/

clock_init()
{
    int i;
    for (i=0; i<MAX_CLOCKS; i++)
        clock_reset(i);
}  /* clock_init */

/*************
 *
 *    wall_time(sec, usec) - It has been sec seconds + usec microseconds
 *        since midnight Jan. 1, 1970 GMT.
 *
 *************/

wall_time(seconds, microseconds)
STAT_TYPE *seconds, *microseconds;
{
    struct timeval t;
    struct timezone tz;

    gettimeofday(&t, &tz);
    *seconds = t.tv_sec;
    *microseconds = t.tv_usec;

}  /* wall_time */

/*************
 *
 *    clock_start(clock_num) - Start or continue timing.
 *
 *        If the clock is already running, a warning message is printed.
 *
 *************/

clock_start(c)
int c;
{
    struct clock *cp;

    cp = &clocks[c];
    if (cp->curr_sec != -1) {
```

```
        fprintf(stderr, "WARNING, clock_start: clock %d already on.\n", c);
        printf("WARNING, clock_start: clock %d already on.\n", c);
        }
    else
        wall_time(&cp->curr_sec, &cp->curr_usec);
} /* clock_start */

/*************
 *
 *   clock_stop(clock_num) - Stop timing and add to accumulated total.
 *
 *      If the clock not running, a warning message is printed.
 *
 *************/

clock_stop(c)
{
    STAT_TYPE sec, usec;
    struct clock *cp;

    cp = &clocks[c];
    if (cp->curr_sec == -1) {
        fprintf(stderr, "WARNING, clock_stop: clock %d already on.\n", c);
        printf("WARNING, clock_stop: clock %d already off.\n", c);
        }
    else {
        wall_time(&sec, &usec);
        cp->accum_sec += sec - cp->curr_sec;
        cp->accum_usec += usec - cp->curr_usec;
        cp->curr_sec = -1;
        cp->curr_usec = -1;
        }
} /* clock_stop */

/*************
 *
 *   STAT_TYPE clock_val(clock_num) - Returns accumulated time in milliseconds.
 *
 *      Clock need not be stopped.
 *
 *************/

STAT_TYPE clock_val(c)
int c;
{
    STAT_TYPE sec, usec, i, j;

    i = (clocks[c].accum_sec * 1000) + (clocks[c].accum_usec / 1000);
    if (clocks[c].curr_sec == -1)
        return(i);
    else {
        wall_time(&sec, &usec);
        j = ((sec - clocks[c].curr_sec) * 1000) +
            ((usec - clocks[c].curr_usec) / 1000);
        return(i+j);
```

```
        }
} /* clock_val */

/*************
 *
 *    clock_reset(clock_num) - Clocks must be reset before being used.
 *
 *************/

clock_reset(c)
int c;
{
    clocks[c].accum_sec = clocks[c].accum_usec = 0;
    clocks[c].curr_sec = clocks[c].curr_usec = -1;
} /* clock_reset */
```

This last file is used by the macros to generate the slave processes.

```
#include <stdio.h>

/*****
 *
 *  These macros create routines to spawn the slave processes
 *
 *****/

ENV

PROCESS_GROUP(one_slave)
  PROCESS_ENTRY(/usr/UMRISCB/users/shaggyk/par/ac.slave,umriscc)
PROCESS_GROUP_END

PROCESS_GROUP(three_slaves)
  PROCESS_ENTRY(/usr/UMRISCB/users/shaggyk/par/ac.slave,umrisca)
  PROCESS_ENTRY(/usr/UMRISCB/users/shaggyk/par/ac.slave,umriscc)
  PROCESS_ENTRY(/usr/UMRISCB/users/shaggyk/par/ac.slave,umriscd)
PROCESS_GROUP_END

PROCESS_GROUP(five_slaves)
  PROCESS_ENTRY(/usr/UMRISCB/users/shaggyk/par/ac.slave,umrisca)
  PROCESS_ENTRY(/usr/UMRISCB/users/shaggyk/par/ac.slave,umriscc)
  PROCESS_ENTRY(/usr/UMRISCB/users/shaggyk/par/ac.slave,umriscd)
  PROCESS_ENTRY(/usr/UMRISCB/users/shaggyk/par/ac.slave,umrisce)
  PROCESS_ENTRY(/usr/UMRISCB/users/shaggyk/par/ac.slave,umriscf)
PROCESS_GROUP_END
```