AN EFFICIENT UNIFICATION ALGORITHM

by

Lewis Denver Baxter

Applied Analysis & Computer Science

July, 1973

Technical Report CS-73-23

ABSTRACT

An efficient algorithm to unify sets of expressions of first order logic is presented. It is shown that previous implementations of the abstract unification algorithm require exponential amounts of resources. The claim of efficiency is partly justified by showing that the space required is linear in relation to the lengths of the inputs. With respect to time, although the previous inefficiencies are eliminated, a precise estimate is difficult to make.

KEYWORDS AND PHRASES

unification, analysis of algorithms, resolution, theorem proving, first order logic, topological sorting, data structures, computational complexity

CR CATEGORIES

3.60 5.21 5.25 5.31

1. INTRODUCTION

The unification computation plays a major role in refutational deduction algorithms and can be compared to the elementary arithmetic operations in procedures of numerical mathematics. Consequently it is important to design the most efficient unification algorithms.

An implementation of the abstract algorithm given in Robinson's pioneer paper [1] using a string representation for expressions is inefficient due to the explicit creation of terms whose length can increase exponentially. Hence such an implementation requires at least an exponential amount of time and space in relation to the length of the input set of expressions.

In [2] Robinson gives an implementation of the unification algorithm using a compact tree-like data structure to represent expressions, which purports to be "very close to maximal efficiency". The data structure certainly economizes on space because instead of creating and copying expressions in the course of applying substitutions, pointers are manipulated; consequently only a linear amount of space is required. However the inherent phenomenon of the abstract algorithm persists: expressions are (implicitly) created whose length is exponential. Although these expressions require only a linear amount of space, the time required to examine them is exponential.

Whereas the above algorithms involve sequential scanning and subsequent substitution, a new algorithm is presented which has a simple transformational approach of a parallel nature. The input set of expressions is transformed into a simpler set which has the same most general unifier (if it exists); in fact a permutation of this set, if unifiable, represents the most general unifier. This set of expressions is unifiable if an associated directed graph has no circuits, or alternatively, if a naturally induced partial order can be topologically sorted. This algorithm also quickly detects failure of unification in the same but broader spirit of [3]. Whereas Stillman uses the concept of "weak substitution" to merely filter through expressions which are "weakly unifiable", this algorithm not only easily detects failure of unification in certain cases but also reports all cases of failure and gives the most general unifier in all cases of success.

The topological sorting phase of the efficient algorithm has been previously analyzed to require linear amounts of resources. Analysis of the transformational phase is discussed and shown to require a linear amount of space. An analysis of the time required is not trivial; one reason is that a problem involving the processing of equivalence relations can be formulated as a unification problem. It is conjectured that the amount of time required is quadratic in relation to the length of the input expressions.

A complete resolution program can be based upon this efficient unification algorithm in which substitutions are never performed but rather denote constraints in the representation of clauses.

2. PRELIMINARIES

The reader is expected to be familiar with the papers [1] and [2], which define the problem of unifying expressions in the context of resolution-oriented theorem provers; much of the nomenclature found there will be used, except that for substitutions. All substitutions in this paper are represented as the implicit product of component substitutions: $\{v \leftarrow t\}$ in which the variable v is to be replaced by the term t. Consequently the general form of a substitution is:

$$\{v_1 \leftarrow t_1\}\ \{v_2 \leftarrow t_2\} \cdots \{v_n \leftarrow t_n\}$$

and will be called a composite substitution. The material on topologically sorting a partial order [4, 2.2.3] will be useful also.

Expressions of first order logic will contain superfluous punctuation for readability (e.g. parentheses, commas and blanks) but to simplify the calculation of the relationship between the length of an input, denoted by L, to an algorithm and the time and space requirements, only the predicate symbol: P, function symbols: f,g,h constant symbols (regarded as nullary function symbols): a,b and variables $x,y,z,u,v,w,x_0,x_1,\ldots,$ whose lengths are taken as unity, will be used.

In the analysis of algorithms the O-notation will be used for approximations [4, 1.2.11.1].

3. ANALYSIS OF PREVIOUS ALGORITHMS

In computing the most general unifier it is assumed that the substitution components are not explicitly "multiplied" together, since otherwise the unification problem would be inherently inefficient. To demonstrate this and to also exemplify the notation used, consider the unification of the set of two literals:

$$\{P(f(x_o,x_o),\ f(x_1,x_1),\ \cdots,\ f(x_{n-1},x_{n-1})),\ P(x_1,x_2,\cdots,x_n)\}\ .$$

The unifier for this set is the composite substitution:

$$\{x_n \leftarrow f(x_{n-1},x_{n-1})\}\ ---\ \{x_2 \leftarrow f(x_1,x_1)\}\ \{x_1 \leftarrow f(x_o,x_o)\}\ .$$

To show this product explicitly, first define the expressions e_i recursively:

$$e_o = x_o\ ;\quad e_{i+1} = f(e_i,e_i)\quad i = 0,1,2,\ldots$$

Then the explicit unifier is:

$$\{x_1 \leftarrow e_1,\ x_2 \leftarrow e_2,\ldots,\ x_n \leftarrow e_n\}\ .$$

To find the relationships between the lengths of these unifiers:

the length of the input, $L = 4n + 2$;
the length of the composite unifier equals $4n = O(L)$;
the length $|e_i|$ of the expression e_i can be shown by induction to be $2^{i+1} - 1$, hence the length of the explicit unifier is:

$$\begin{aligned}\sum_{i=1}^{n} (1 + |e_i|) &= \sum_{i=1}^{n} 2^{i+1}\\ &= 4\ (2^n-1)\\ &= O(2^L)\ .\end{aligned}$$

There are two common implementations of the original abstract unification algorithm [1], according to the data structure used to represent expressions.

In the implementation which uses a string representation [1] the unification of:

$$\{P(f(x_o,x_o),\ f(x_1,x_1),\ldots,\ f(x_{n-1},x_{n-1}),\ P(x_1,x_2,\ldots,\ x_n)\}$$

requires the creation of terms having exponential length. Using the earlier definition of the expressions e_i, after applying the substitutions:

$$\{x_1 \leftarrow e_1\},\ \{x_2 \leftarrow e_2\},\ldots,\ \{x_{n-1} \leftarrow e_{n-1}\}\ ,$$

at the final stage of the algorithm the terms x_n and e_n are examined. To determine if the variable x_n occurs in the term e_n, it is necessary to search e_n. Hence the time required for the algorithm is bounded below by the time required to examine e_n, which is proportional to its length. Also, since all expressions are explicitly represented by strings, the space required for the algorithm is bounded below by the space required to store the expression e_n. As already seen, e_n has exponential length thus both resources are at least exponential for this implementation.

In [2] an implementation is presented in which expressions are represented by trees in which common subtrees are identified, thus e_n has the representation:

$$\begin{tikzcd} f \arrow[r, bend left] \arrow[r, bend right] & f \arrow[r, bend left] \arrow[r, bend right] & \text{- - -} \arrow[r, bend left] \arrow[r, bend right] & f \arrow[r, bend left] \arrow[r, bend right] & x_o \end{tikzcd}$$

in which the pointers from the binary function f refer to its two arguments which in this case are identical expressions.

Although the representation of e_n requires only linear space, the time taken to search e_n for an occurrence of x_n remains exponential due to the essential recursive nature of the examination.

Actually, the space required to store expressions is quadratic due to the two-dimensional table, "args", whose number of rows is proportional to the length of the input expressions and whose number of columns is the maximum "arity" of the input functions. However, by using a linked-list representation for "args" the quadratic bound becomes linear. Also note that the execution of the recursive search function "occur" implicitly requires additional space in the form of a pushdown stack which however is only linear. In practice, the space requirements are not critical since most storage can be released at the end of the algorithm, which would appear as a subroutine in a theorem prover.

4. DESCRIPTION OF THE EFFICIENT ALGORITHM

The efficient algorithm which is now presented determines if a set of pairs of expressions:

$$S = \{ \{r_1,s_1\},\ldots,\{r_n,s_n\} \}$$

if unifiable and if need be computes the most general unifier of S. The algorithm can be readily applied to determining the most general unifier of several (m) literals:

$$\{P(t_{11},\ldots,t_{1n}),\ldots,P(t_{m1},\ldots,t_{mn})\}$$

since that problem is equivalent to unifying the set of m-1 pairs of literals:

$$\{ \{P(t_{11},\ldots,t_{1n}),\ P(t_{21},\ldots,t_{2n})\},\ldots,\{P(t_{11},\ldots,t_{1n}),\ P(t_{m1},\ldots,t_{mn})\} \}.$$

First an informal description of the algorithm with illustrating examples is given. The algorithm consists of two separate phases: a <u>transformational phase</u> followed by a <u>sorting phase</u>. In the first phase the set S is transformed, according to several rules, into another set U of ordered pairs of expressions:

$$U = \{<v_1,t_1>,\ldots,<v_m,t_m>\}$$

in which the variables v_i are distinct. The first element of the ordered pair <v,t> will be called the <u>superseded</u> variable. Each rule preserves unifiability so that U is unifiable iff S is unifiable. During the algorithm a pair from S is either transferred to U or is replaced by some other pairs. Initially U is empty and the algorithm proceeds iteratively by selecting an arbitrary pair from S until either S becomes empty or failure of unification is detected. According to the form of a particular pair of expressions chosen from S one of the following rules is applied:

R1. The pair {t,t}, where t is any term, is deleted from S.

R2. If the pair is $\{f(s_1,\ldots,s_m),\ g(t_1,\ldots,t_n)\}$ then if the function symbols f and g differ, unification fails otherwise replace the pair by the m (=n) pairs: $\{s_1,t_1\},\ldots,\{s_m,t_m\}$.

R3. If the pair is {v,t} where v is a variable and t is a term (possibly also a variable) and if neither v nor t appears as a superseded variable of U then add <v,t> to U.

R4. If the pair is {v,t} where v is a variable which appears as a superseded variable in the ordered pair <v,t'> of U then replace {v,t} by {t,t'} in S.

Note that the rules R3 and R4 include the case in which the pair from S consists of two variables.

Eg 1. S = { {f(h(a,w,a),x),y},{z,g(y,a)},{g(f(u,x),w),z} }.
The sets S and U are shown at the beginning of each iteration with the chosen pair of S underlined.

Rule	S ; U
R3	$\{f(h(a,w,a),x),y\}\ \underline{\{z,g(y,a)\}}\ \{g(f(u,x),w),z\}\ ;$
R4	$\{f(h(a,w,a),x),y\}\ \underline{\{g(f(u,x),w),z\}}\ ;\ <z,g(y,a)>$
R3	$\underline{\{f(h(a,w,a),x),y\}}\ \{g(f(u,x),w),\ g(y,a)\}\ ;\ <z,g(y,a)>$
R2	$\underline{\{g(f(u,x),w),g(y,a)\}};\ <z,g(y,a)>\ <y,f(h(a,w,a),x)>$
R3	$\{f(u,x),y\}\ \underline{\{w,a\}}\ ;\ <z,g(y,a)>\ <y,f(h(a,w,a),x)>$
R4	$\underline{\{f(u,x),y\}};\ <z,g(y,a)>\ <y,f(h(a,w,a),x)>\ <w,a>$
R2	$\underline{\{f(u,x),f(h(a,w,a),x)\}}\ ;\ <z,g(y,a)>\ <y,f(h(a,w,a),x)>\ <w,a>$
R1	$\{u,h(a,w,a)\}\ \underline{\{x,x\}}\ ;\ <z,g(y,a)>\ <y,f(h(a,w,a),x)>\ <w,a>$
R3	$\underline{\{u,h(a,w,a)\}}\ ;\ <z,g(y,a)>\ <y,f(h(a,w,a),x)>\ <w,a>$
	$;\ <z,g(y,a)>\ <y,f(h(a,w,a),x)>\ <w,a>\ <u,h(a,w,a)>$

The resulting set U equals:

$$\{<z,g(y,a)>,\ <y,f(h(a,w,a),x)>,\ <w,a>,\ <u,h(a,w,a)>\}\ .$$

In the ordering phase, determination if U is unifiable requires the recognition of a circuit in an associated directed graph, G, which reflects the relationship between the variables in U. If U = $\{<v_1,t_1>,\ldots,<v_n,t_n>\}$ then G has as nodes the superseded variables $\{v_1,\ldots,v_n\}$ and directed edges $v_i \leftarrow v_j$ for each variable v_j occurring in the term t_i. U is unifiable iff G contains no circuits.

Eg 2. From Eg 1, U has the associated graph:

$$\begin{tikzcd} z & y \arrow[l] & w \arrow[l] \arrow[r] & u \end{tikzcd}$$

which has no circuits hence S is unifiable. (Note that x is not a superseded variable of U.)

Eg 3. U = {<x,f(z,v,w)>, <y,g(u,w)>, <z,h(y,w)>, <u,g(w,v)>, <v,h(z,a)>} has the associated graph:

$$\begin{tikzcd}
& z \arrow[dl] \arrow[dd] & y \arrow[l] \\
x & & \\
& v \arrow[ul] \arrow[r] & u \arrow[uu]
\end{tikzcd}$$

which has a circuit: $z \leftarrow y \leftarrow u \leftarrow v \leftarrow z$, hence U is not unifiable.

Eg 4. U = {<x,f(x,b,x)>} has the associated graph:

$$\begin{tikzcd}
x \arrow[loop above]
\end{tikzcd}$$

which has a circuit, hence U is not unifiable.

Eg 5. Note that due to rule R4, circuits formed from variables will not occur. If S = { {x,y},{y,z}, {z,x} } , which is unifiable, then after the transformational phase U = {<x,y>, <y,z>} and U $\neq$ {<x,y>, <y,z>, <z,x>} whose associated graph has a circuit.

In general, if U is unifiable then some permutation of it gives the most general unifier: <v,t> represents the substitution component {$v \leftarrow t$} of the composite unifier.

The problem of determining if a directed graph has a circuit is equivalent to that of determining if the naturally induced partial order can be topologically sorted. The topological sort which also gives the order of the component substitutions, has a well known linear algorithm given in Knuth [4, 2.2.3].

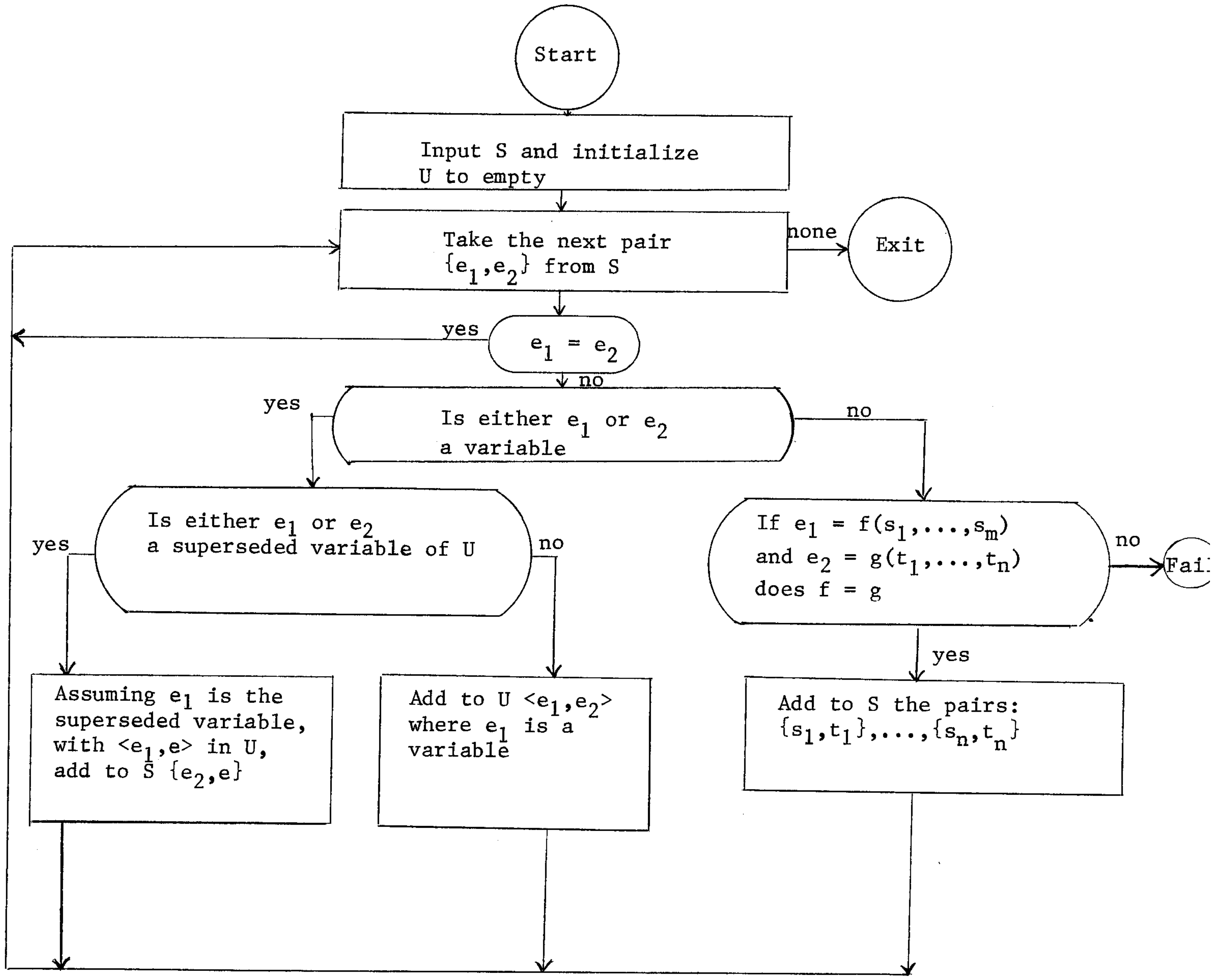
Eg 6. Using Eg 2 the partial order naturally induced from the graph is $\{z \leftarrow y,\ y \leftarrow w,\ u \leftarrow w\}$ where $v_1 \leftarrow v_2$ denotes "v_1 precedes v_2". This can be topologically sorted into the ordered list (z y u w), in which, if $v_1 \leftarrow v_2$ then v_1 precedes v_2 in the sorted list. Hence the composite unifier is:

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

in which the order of the component substitutions corresponds to that of the associated variables in the topological sort.

5. FLOWCHART AND DATA STRUCTURES

A more formal flowchart to describe the transformational phase of the algorithm is now given:


Start
Input S and initialize U to empty
Take the next pair $\{e_1,e_2\}$ from S
none
Exit
yes
$e_1 = e_2$
no
yes
Is either e_1 or e_2 a variable
no
yes
Is either e_1 or e_2 a superseded variable of U
no
If $e_1 = f(s_1,\ldots,s_m)$ and $e_2 = g(t_1,\ldots,t_n)$ does $f = g$
no
Fai
yes
Assuming e_1 is the superseded variable, with $<e_1,e>$ in U, add to S $\{e_2,e\}$
Add to U $<e_1,e_2>$ where e_1 is a variable
Add to S the pairs: $\{s_1,t_1\},\ldots,\{s_n,t_n\}$

In the algorithm, the pair chosen from S is arbitrary; if S is represented as a linked list stack, pairs are deleted from and inserted at one end; if S is represented as a queue, pairs are taken from one end and added to the other. By using heuristics it may be possible to choose pairs which will quickly lead to a failure if that is inevitable.

If expressions are represented by strings then the above algorithm is not linear, for consider the unification of $\{s_n,t_n\}$ where the expressions s_i and t_i are recursively defined:

$$s_o = x \ , \ t_o = y \ ; \ s_{i+1} = f(s_i,t_i) \ , \ t_{i+1} = f(t_i,s_i) \quad i = 0,1,2,\ldots$$

The set S during the algorithm contains in all: 1 copy of $\{s_n,t_n\}$, 2 copies of $\{s_{n-1},t_{n-1}\},\ldots,2^n$ copies of $\{s_o,t_o\}$, the total length of which is:

$$L' = \sum_{i=o}^{n} 2^{n-i} (|s_i|+|t_i|) = \sum_{i=o}^{n} 2^{n-i}\cdot 2\cdot(2^{i+1}-1)$$

$$= 2 \sum_{i=o}^{n} (2^{n+1}-2^{n-i})$$

$$= 2 \ (\ (n+1)2^{n+1}-(2^{n+1}-1).$$

Now the length of $\{s_n,t_n\}$,

$$L = |s_n|+|t_n| = 2 \ (2^{n+1}-1)$$

hence $L' = 0(L \log L)$, consequently the time and space required to copy these expressions is not linear.

If however, expressions are represented by the compact tree-like data structure the above inefficiency is eliminated because only pointers rather than expressions need be manipulated. The above flowchart can then be reinterpreted to reflect the fact that only references to the expressions are manipulated.

This is illustrated by using the "table" representation of expressions described by Robinson [2]. The k-th row of this table represents an expression, expression [k]; SYMBOL [k] is the first symbol of that expression and if this is a function symbol then ARGS [k,i] (efficiently represented as a linked list) refers to the i-th argument of the function. VBLE [k] indicates if the expression is a variable and if so, SUBST [k] if non-zero refers to the pair:

<expression [k], expression[SUBST [k]]>

in U. S, initially a set references, is represented by a linked list data structure.

Eg 7. Using Eg 1 the table representing the sets S and U is finally:

k	SYMBOL	ARGS	VBLE	SUBST
1	f	2 5		
2	h	3 4 3		
3	a			
4	w		true	3
5	x		true	
6	y		true	1
7	z		true	8
8	g	6 3		
9	g	10 4		
10	f	11 5		
11	u		true	2

Initially the SUBST column, representing U, is empty and

S = { {1,6}, {7,8}, {9,7} }; finally

U = {<7,8>, <6,1>, <4,3>, <11,2>}.

6. ANALYSIS OF TIME REQUIREMENTS

The time required for the transformational phase is closely related to the number of times the four rules are applied. It appears certain that this is not linear in relation to the length of the input, due to the repeated processing of pairs from S which consists of two variables. In fact, the problem of processing equivalence relations [4, 2.3.3] can be formulated as a unification problem: if the set of equivalence relations is $\{x_i \equiv y_i \mid i=1,\ldots,n\}$ then by identifying x_i and y_i as variables, to determine if arbitrary elements x_j and y_k are in the same equivalence class the following set of expressions can be considered:

$$S = \{ \{x_1,y_1\},\ldots,\{x_n,y_n\},\ \{x_j,f(y_k)\} \} .$$

S is unifiable iff x_j and y_k are not in the same equivalence class. The efficient unification algorithm probably does not process the sets $\{x_i,y_i\}$ in the most efficient manner. An efficient algorithm [5] which processes equivalence relations by representing equivalence classes by dynamically varying trees, gives a time bound of $O(n \log n)$. It is therefore conjectured that the efficient unification algorithm operates in quadratic time although this is not justified in this paper.

7. CONCLUSION

The unification algorithm presented here processes expressions in a natural parallel manner and consequently eliminates the inefficiencies of earlier algorithms. This new approach raises possibilities for implementing the algorithm on parallel processors, however, a more radical venture is to design a complete theorem prover using the algorithm as a basis. The representation of substitutions as composite substitutions also facilitates the representation of clauses as constraints in the manner of [6].

Several theorem provers have been implemented in which substitutions are not actually performed [7]; this idea together with the use of constraints suggests an efficient implementation of a theorem prover which is dominated by the manipulation of pointers.

This paper suggests further areas of research:

(1) analysis of the time required for the efficient unification algorithm

(2) embedding the efficient unification algorithm in a refutational deduction system using constrained resolution

(3) extension of the efficient unification algorithm to higher order logic.

ACKNOWLEDGEMENT

I am indebted to Prof. T. Pietrzykowski who suggested several improvements to previous versions of this paper and who also discussed further areas of research related to this paper.

REFERENCES

[1] Robinson, J.A. (1965)
A machine-oriented logic based on the resolution principle, JACM 12 23-41.

[2] Robinson, J.A. (1970)
Computational logic: the unification computation, Machine Intelligence 6 63-72.

[3] Stillman, R.B. (1972)
The concept of weak substitution in theorem proving, Ph.D. thesis, Syracuse University.

[4] Knuth, D.E. (1968)
The Art of Computer Programming, Vol. I, Fundamental Algorithms, Addison-Wesley.

[5] Fischer, M. (1972)
Efficiency of equivalence algorithms, in complexity of computer computations, R.E. Miller and J.W. Thatcher (editors), Plenum Press, New York.

[6] Huet, G.P. (1972)
Constrained resolution: a complete method for higher order logic,
Report 1117, Jennings Computing Center, Case Western Reserve University.

[7] Boyer, R.S. and Moore, J.S. (1971)
The sharing of structure in resolution programs, Metamathematics Unit, University of Edinburgh, (To appear in Machine Intelligence 7).