A PRACTICALLY LINEAR UNIFICATION ALGORITHM

by

Lewis Denver Baxter

Research Report CS-76-13

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

February, 1976

ABSTRACT

An algorithm which unifies first-order expressions is presented. The algorithm is proved correct and also the time taken is shown to be O(nG(n)) where G is practically constant. This is an improvement over previous algorithms which required exponential or quadratic time.

KEYWORDS AND PHRASES

computational logic, unification, analysis of algorithms, theorem-proving, topological sorting

1. INTRODUCTION

The unification problem is to determine, when given two expressions containing variables, whether or not there exists a substitution of expressions for those variables which applied to the two expressions makes them equivalent. We consider only first-order expressions.

Unification, under the name "L.C.M." (in analogy to the arithmetic property of least common multiple), was mentioned as early as 1921 by the logician, Post [7]. He was interested in the matching problems associated with generalizations of classical axiomatizations of propositional calculus. Again in 1955 a computer program called the "Logic Theorist" [6], which attempted to prove theorems of propositional calculus by heuristic methods, used a matching process very similar to unification. However it was not until the landmark paper of Robinson in 1965 [8] that unification gained an official title and was presented as an algorithm.

Any computation which makes inferences from logical expressions must inevitably incorporate a unification algorithm. This is the case in theorem-proving systems, whether they be resolution-oriented or of the natural-deduction type, and in programs which apply theorem-proving techniques such as in program analysis/synthesis, deductive question-answering systems and robot technology. Unification also provides the mechanism involved in pattern-directed invocation of subroutines which are featured in some programming languages of artificial intelligence, such as PROLOG [11]. Hence we see the need to develop very efficient unification algorithms.

Robinson's original abstract algorithm requires exponential time. An improved version which uses a tree-like data structure to represent

expressions was presented in [9]. However a paradoxical situation arose: expressions which require linear space to store in the form of a tree with shared subexpressions may now require exponential time for the simple operation of determining if a variable occurs in the expression. This oversight was remedied by Venturini Zilli [12] whose algorithm also maintained a list of variables which an expression contained. It was proved that this algorithm requires quadratic time.

A partial breakthrough was achieved in [2] where the operation of determining if a variable occurs in an expression is delayed until the end. The first of two stages in the algorithm transforms sets of expressions into simpler sets which is now an easy instance of the unification problem. This easy case can be solved in linear time during the second sorting stage by determining if an associated directed graph contains a circuit.

In this paper the algorithm in [2] is improved upon by speeding up the transformational stage. The basic operations in this stage manipulate sets of expressions: two sets are merged and the set containing an expression is found. Using well-known techniques for these operations, in which sets are represented as balanced trees, an upper bound for the time is found to be of the order of $nG(n)$ where G is "practically" constant. For example, whenever n is less than 2^{65536}, $G(n)$ is at most 5.

In section 2 of this paper we define our notation. In section 3 we describe two strategies of designing unification algorithms and explain why one of these, of which our algorithm is an example, is superior than the other. We also describe our algorithm abstractly and then describe the data structures

used to obtain an efficient implementation. Section 4 contains a flowchart describing our abstract algorithm and a collection of structured programs which implements it. In Section 5 we prove that both the transformational and sorting stages of our algorithm are correct. In Section 6 we perform an analysis of the time required by the algorithm.

2. PRELIMINARIES

Here we introduce our notation and conventions.

The reader should refer to [8] for a formal presentation of expressions and substitutions. Briefly an expression is either a variable or an n-ary (constant) function-symbol followed by n expressions. We define a term as an expression which is not a variable.

A substitution is a finite set of ordered pairs $\{\langle v_1,e_1\rangle,\ldots,\langle v_n,e_n\rangle\}$ in which the v_i's are distinct variables and the e_i's are expressions. We write such a substitution as $\{v_1 \leftarrow e_1,\ldots,v_n \leftarrow e_n\}$. A substitution can be considered to be a function which maps variables into expressions; this function can then be extended to expressions. We will use the lower-case Greek letters ρ, σ and τ to denote substitutions. Prefix notation is consistently used when writing substitutions.

In this paper all substitutions are represented as the implicit product of component substitutions. If this product is to be "multiplied" out then the unification problem will be inherently inefficient, as explained in [2]. Hence the meaning of the product of substitutions:

$$\{v_n \leftarrow e_n\} \ldots \{v_1 \leftarrow e_1\}$$

is to first apply $\{v_1 \leftarrow e_1\}$ by substituting every occurrence of the variable v_1 by the expression e_1, then apply $\{v_2 \leftarrow e_n\},\ldots$, and finally apply $\{v_n \leftarrow e_n\}$.

The unification problem is to decide, given two expressions e_1 and e_2, if there exists a substitution σ such that $\sigma(e_1) = \sigma(e_2)$. If such a unifying substitution exists we would like to find the most general such unifier.

We now define some concepts.

The substitution σ <u>unifies</u> the set of expressions, $\{e_1,\ldots,e_n\}$ if and only if (iff) $\sigma(e_1) = \ldots = \sigma(e_n)$.

σ unifies the set S containing sets of expressions iff $T \in S \rightarrow \sigma$ unifies T.

In the next two definitions X is either a set of expressions or a collection of sets of expressions.

σ is a <u>mgu</u> (most general unifier) of X iff σ unifies X and $\forall\rho[\rho$ unifies $X \rightarrow \exists\tau(\rho = \tau\sigma)]$.

X is <u>unifiable</u> iff $\exists\sigma(\sigma$ unifies X).

The time taken during an algorithm is relative to the length of, or number of symbols in, expressions. In computing this length only variables (denoted by lower-case letters) and function-symbols (denoted by upper-case letters) are counted. Formally:

$$\text{length}\,[f(e_1,\ldots,e_n)] = 1 + \sum_{i=1}^{n} \text{length}\,[e_i]$$

where either f is a variable and n=0 or f is a function-symbol and $e_1,\ldots,e_n$ are its $n \geq 0$ arguments.

In the analysis of algorithms the O-notation will be used. An algorithm requires O(f(n)) time relative to the length of the input, n, if there exists a constant c such that at most c.f(n) steps are taken by the standard model of a computing device: the Random Access Machine [1].

3. DESCRIPTION OF THE ALGORITHM

So far, all unification algorithms which have been presented can be characterized by two overall strategies: $STRATEGY_1$ and $STRATEGY_2$. We will explain why $STRATEGY_1$ requires at least quadratic time and will justify our choice of $STRATEGY_2$.

In $STRATEGY_1$ whenever a pair {v,e} is to be unified it is determined if the variable v occurs in the expression e; if so unification fails else the substitution $\{v \leftarrow e\}$ is applied. Robinson's original abstract algorithm used this strategy and was inefficient since the substitution was explicitly applied in a character-string representation of expressions. This strategy also characterizes the algorithms presented in [9,12].

$STRATEGY_2$ consists of two stages: a transformational stage in which sets of expressions are simplified, followed by a sorting stage in which, effectively, a directed graph is examined for the presence of a circuit. If one exists unification fails otherwise the graph is sorted to find the unifying substitution. The algorithm in [2] and in this paper use this strategy.

The graphical perspective is important since the analysis of graphical algorithms justifies why $STRATEGY_2$ is superior. Furthermore $STRATEGY_2$ requires only linear time for the special case of unification in which all variables belong to exactly one set of expressions. Here the transformational stage is quickly applied and the sorting stage, which always requires only linear time, is performed. We will construct such an easy case of unification from a directed graph and by comparing $STRATEGY_1$ with the analogous graphical method we will show why $STRATEGY_1$ is inefficient.

Given a directed graph we can construct sets of expressions such that the graph has a circuit iff the set is not unifiable. This construction is done by associating with each vertex v of the graph a distinct variable v'. If $v \rightarrow v_1,\ldots,v \rightarrow v_n$ are all the edges emanating from v then we construct the pair of expressions: $\{v',F_n(v_1',\ldots,v_n')\}$ where F_n is an n-ary function-symbol. All such pairs constructed from all vertices comprise the collection of sets of expressions.

This constructed set is a special case mentioned earlier, hence takes linear time using $STRATEGY_2$. For this case we are convinced that $STRATEGY_1$ will take quadratic time for the following reasons. If the constructed set is processed using $STRATEGY_1$ then effectively an algorithm belonging to this class operates "on-line", that is, referring to the graphical analogy, the graph is constructed edge by edge. After each edge is added to the graph it is determined whether there is a circuit due to the addition of that edge. (This corresponds to the checking of a variable occurring in an expression). The next edge is added, etc., so that the checking for circuits occurs continually. It is conjectured that no algorithm which continually checks for circuits in the manner of $STRATEGY_1$ can operate in linear time, even for this special case. If such an algorithm existed we would have a circuit-detecting algorithm which operates in "real-time", that is, the amount of time spent between adding each successive edge is constant. No such real-time algorithm is known and its non-existence is supported by results on the transitive-closure algorithm [5]. Such a real-time algorithm must effectively update the transitive-closure graph (which is induced by the relation: "path between") in constant time. If this were possible we would have a transitive-closure algorithm more efficient than the known best.

This would also be the case if $STRATEGY_1$ was better than quadratic. Having justified why $STRATEGY_2$ is probably superior, we will now present a new algorithm of that class which retains much of the flavour of [2]. Since the sorting stage has a well-known linear algorithm [4] we will concentrate on the transformational stage.

Transformational Stage

This stage inputs a set of pairs of expressions to be unified, S_I, and transforms them into a collection of sets of expressions, F_0. Here unification may fail due only to the attempted unification of a pair of terms with different function-symbols. The two main sets used in the transformational stage are S, a set of pairs of expressions and F, a collection of sets of expressions. F always has two special properties: all the terms (if any) in each set of F begin with the same function-symbol and each subexpression of the input set, S_I, belongs to exactly one set of F. Initially S is S_I and F consists of all the subexpressions in S_I, each in a set of its own. Finally S is empty and F is the output set, F_0. The relationship between S and F is given by the following invariant property:

$$\exists\, \sigma \ (\sigma \text{ unifies S and } \sigma \text{ unifies F}).$$

Intuitively consider that we are unifying S subject to the constraints given by F.

We now give an outline of our method. This will be expanded in more detail later in the form of a flowchart and programs.

repeat until S is empty:

begin delete any pair $\{e_1,e_2\}$ from S;

let $e_1 \in T_1$ and $e_2 \in T_2$ where $T_1,T_2 \in F$;

if T_1 contains some term $f'(e'_1,\ldots,e'_n)$ and

T_2 contains some term $f''(e''_1,\ldots,e''_m)$

then if $f' \neq f''$

then UNIFICATION FAILS

else add to S the pairs $\{e'_1,e''_1\},\ldots,\{e'_n,e''_n\}$;

merge T_1 and T_2, that is replace T_1 and T_2 by $T_1 \cup T_2$;

end.

The justification for the merging is, where $e_1 \in T_1$ and $e_2 \in T_2$:

σ unifies $\{e_1,e_2\}$ & σ unifies T_2 iff σ unifies $T_1 \cup T_2$.

Also σ unifies $\{f'(e'_1,\ldots,e'_n),f''(e''_1,\ldots,e''_m)\}$

iff $f'=f''$ & $m=n$ & σ unifies $\{\{e'_1,e''_1\},\ldots,\{e'_n,e''_n\}\}$.

Note that the two special properties of F are maintained. Each subexpression of S_I belongs to exactly one set of F since initially this is trivially true and merging preserves this property. Also the test f'=f" ensures that all terms belonging to a set of F begin with the same function-symbol.

The important operations of this transformational stage are:

FIND which set of F an expression belongs to and MERGE two sets of expressions.

To efficiently perform these basic operations we represent F as a forest of balanced trees. Each set of F is represented as a tree, each

vertex of which refers to an expression. Since we must know if a set contains a term, the root of a tree points to some arbitrary term within that tree. The root of a tree is effectively the name of the set which is represented by the tree. Figure 3.1 illustrates a forest.

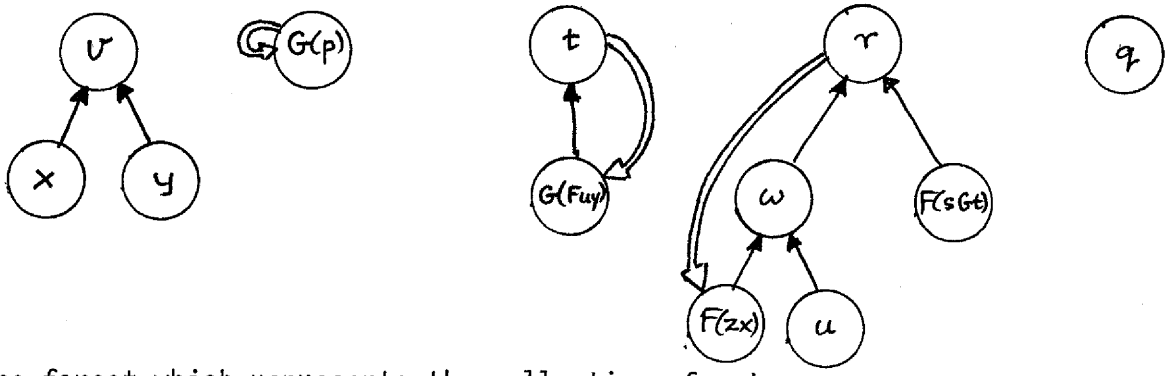
To find which set an expression belongs to we traverse a path from the vertex of the tree corresponding to the expression towards the root. We then collapse this path directly onto the root, that is, each vertex of this path now points directly to the root. Figure 3.2 illustrates such an example.

To merge two sets we balance the representing sets by making the "lighter" tree a subtree of the "heavier" tree, where the comparatives refer to the number of vertices in the trees. In the case when the "heavy" subtree contains only variables and the "light" subtree contains some term we have to ensure that the new root points to some term. If both sets contain terms then we must put the appropriate arguments into S by examining the terms pointed to by the roots. Figure 3.3 gives an example of merging. Initially F is the collection F_I, of all subexpressions in the input set S_I, hence is represented by a "newly sown forest" in which each tree of the forest contains only one vertex.

The set of pairs of expressions, S, is represented as a stack; to choose a pair from S we simply "pop" off this stack the top two elements which refer to expressions.

The expressions themselves are represented by trees in which all common variables are shared. That is, different occurrences of the same variable are represented by different pointers to the same vertex of the

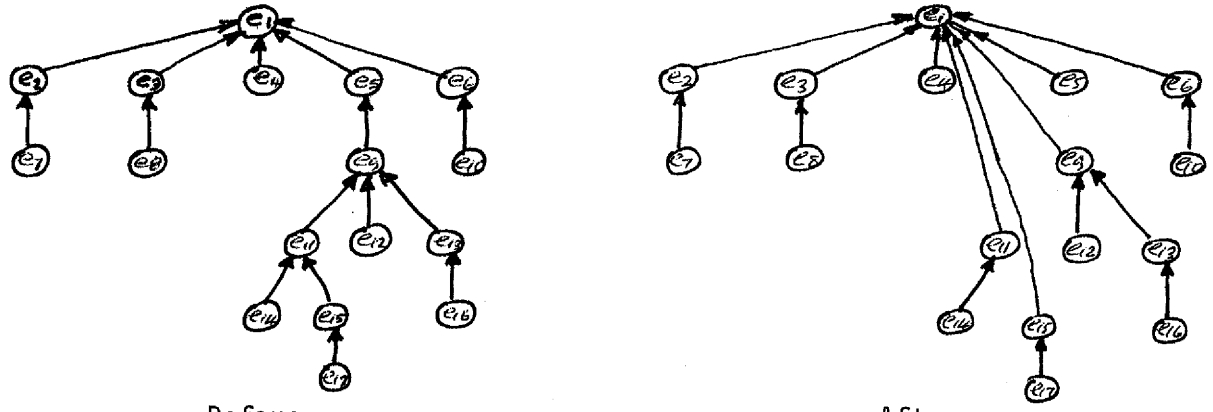
Figure 3.1:



The forest which represents the collection of sets:

{ {x,y,v}, {G(p)}, {G(F(u,y)),t}, {F(z,x),u,w,F(s,G(t)),r}, {q} }

Figure 3.2:

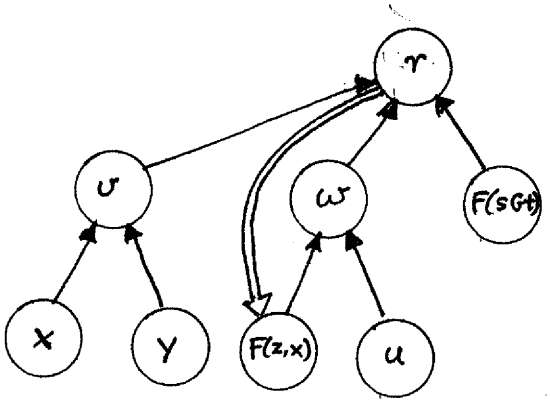


Before

After

Both trees represent $\{e_1, e_2, \ldots, e_{17}\}$, before and after FINDING the set which contains the expression, e_{15}. The vertices e_{15}, e_{11}, e_9 and e_5 on the path from e_{15} to the root are collapsed directly onto the root.

Figure 3.3:



This tree is obtained after MERGing the first and fourth trees of Figure 3.1. It represents the set {x,y,v,F(z,x),u,w,F(s,G(t)),r}

tree. Figure 3.4 illustrates both the representation of expressions and of S as a stack. Each expression is in fact a pointer to a list of its attributes: SYMBOL, VARIABLE, ARGLIST; PARENT, TERM and COUNT. The first three represent the expression and hence never change and the last three attributes are used in the representation of F as a forest of trees.

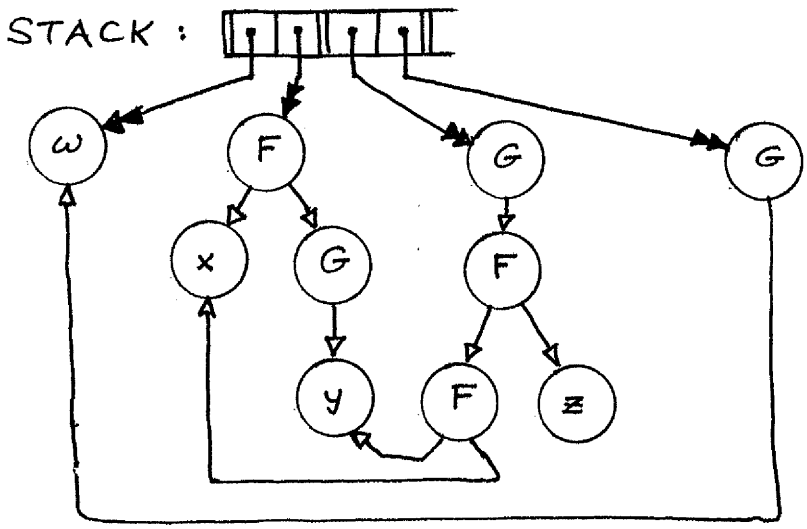
SYMBOL is the first character of an expression, VARIABLE indicates whether the expression is a variable or a term and if it is a term, ARGLIST points to a list of its arguments. In the forest representation of sets of expressions PARENT indicates the next vertex towards the root of a tree. If PARENT is null then the expression is at the root in which case TERM indicates if the particular set also contains a term. Also COUNT tells us the number of vertices in the tree and is used to balance trees during a merge operation. Initially for all subexpressions PARENT is null, COUNT is 1 and for terms, TERM points to itself otherwise for variables TERM is null.

Sorting Stage

In this stage we construct from the output set F_0 of the transformational stage a directed graph whose size is linear relative to the length of S_I. We attempt to topologically sort this graph, that is embed the vertices in a linear order. If we cannot do this then the graph contains a circuit and F_0 is therefore not unifiable. If we can then the sorted graph indicates the most general unifier of F_0.

In theory, each vertex of the directed graph corresponds to each set in F_0. Given a set T of F_0, we choose, if possible, only one term from it. Let $v_1,\ldots,v_n$ be all the variables which occur in the term where

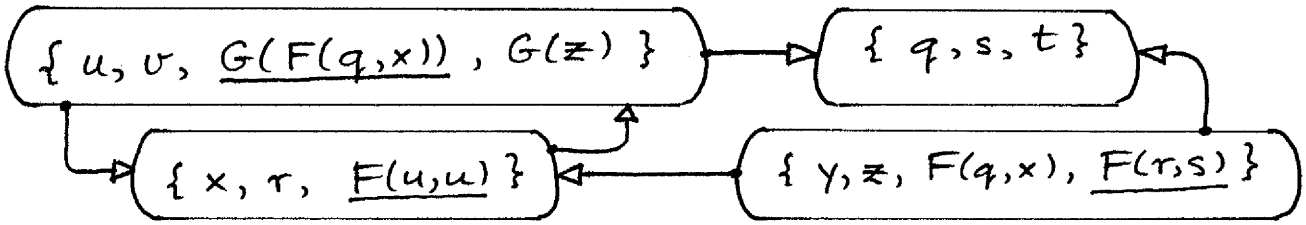
Figure 3.4:



STACK represents S, the set of pairs of expressions:

{ {w,F(x,G(y))}, {G(F(F(y,x),z)), G(w)} }

Figure 3.5:



The directed graph associated with the collection:

{ {u,v,G(F(q,x)),G(z)}, {q,s,t}, {x,r,F(u,u)}, {y,z,F(q,x), F(r,s)} }.

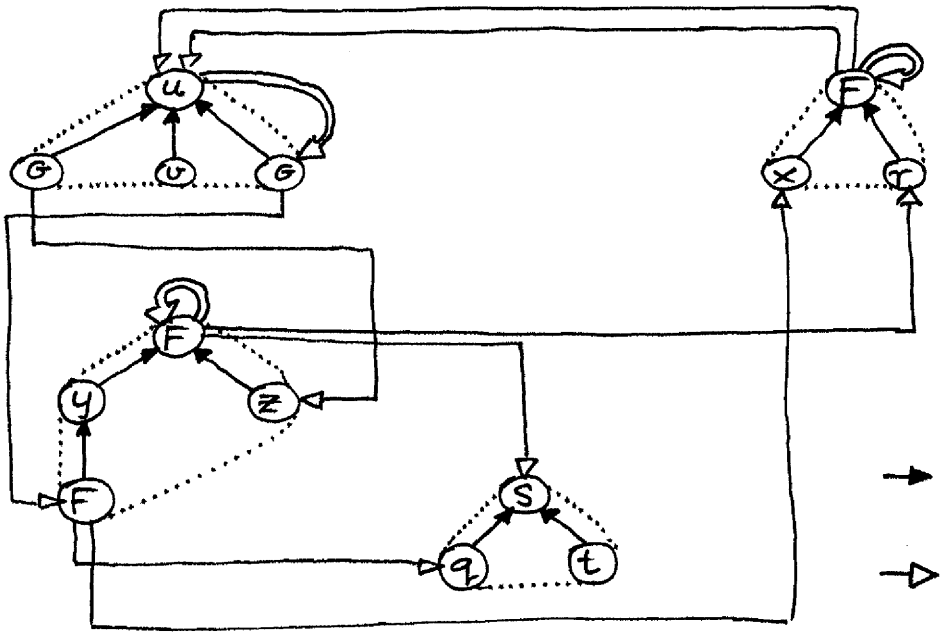
The underlined terms denote particular representatives of the sets, from which the directed graph is constructed.

variable v_i belongs to the set T_i of F_0. We then construct directed edges: $T \rightarrow T_i$ from the vertex corresponding to T toward the vertex corresponding to T_i. Figure 3.5 illustrates this construction.

In practice, however, we must construct a similar directed graph directly from the forest representation of F_0. Each vertex in this directed graph corresponds to a variable occurring in F_0 or to a root of a tree in F_0. If in the tree representation of a set T belonging to F_0, vertex v, corresponding to a variable, has root r then we construct the directed edge: $v \rightarrow r$. If the root of a tree T is r and if a term belongs to T which contains variables $v_1,\ldots,v_n$ then we construct the edges: $r \rightarrow v_i$. The directed graph is constructed by examining each such tree in F_0. The number of vertices and edges in this graph is linear relative to the length of S_I. A detailed algorithm is given in the next section to construct from F_0 the directed graph which is then input to the topological sorting algorithm.

If the topological sorting algorithm detects a circuit in the directed graph then we exit with unification failing. Otherwise a linear sort of the vertices will be output, from which we can easily construct the most general unifier. See Figures 3.6 to 3.9 for examples illustrating these constructions.

Figure 3.6:



Arrows

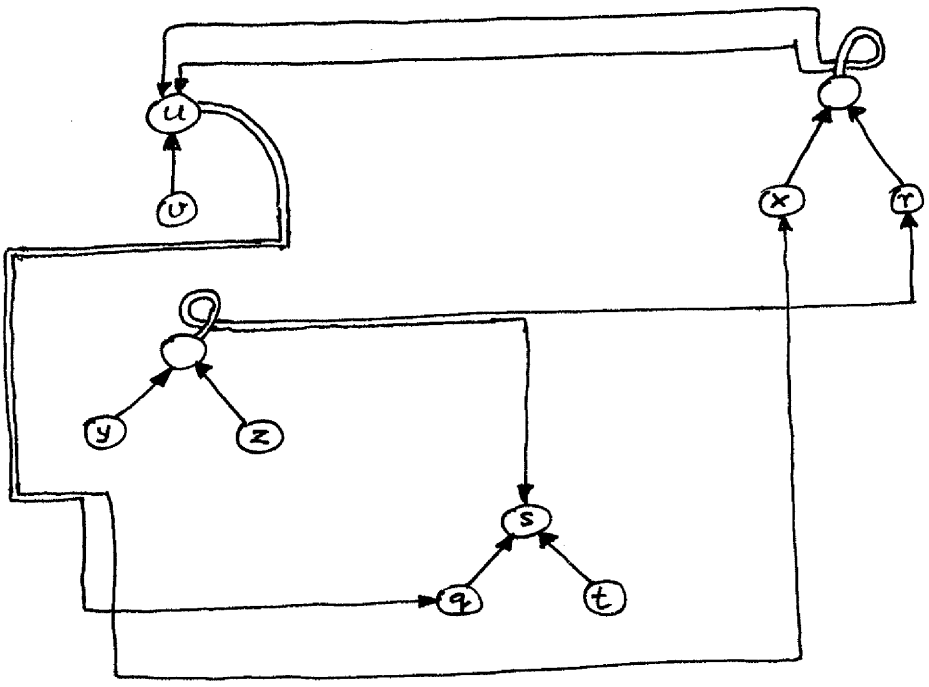
→ : links vertices in the trees.

⇀ : indicates the arguments of a term.

⇨ : points to a particular term in a tree.

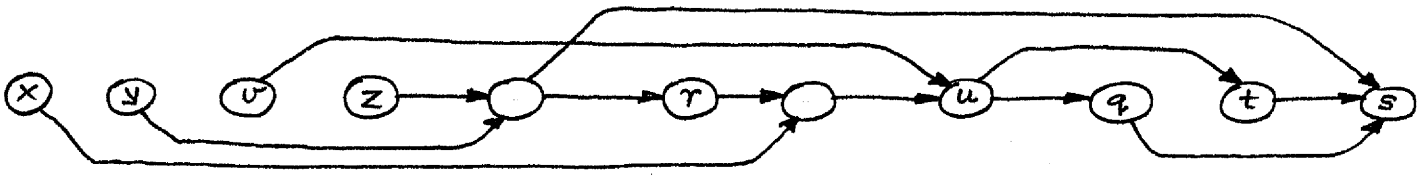
The forest representation of the collection of Figure 3.5.

Figure 3.7:



The directed graph associated with the collection of Figure 3.5, constructed from the forest representation of Figure 3.6.

Figure 3.8:



The directed graph in Figure 3.7 contains one circuit ⓤ → ⓧ → ○ → ⓤ. If we replace the edge ⓤ → ⓧ by ⓤ → ⊕ then the new graph can be topologically sorted as indicated here.

Figure 3.9:

The most general unifier for the collection:

$$\{\{u,v,G(F(q,t)),\ G(z)\},\ \{q,s,t\},\{x,r,F(u,u)\},\{y,z,F(q,t),F(r,s)\}\}$$

induced by the topological sort in Figure 3.8 is $\sigma_8\ \sigma_7 \cdots \sigma_1$

where $\sigma_1 = \{x \leftarrow F(u,u)\}$

$\sigma_2 = \{y \leftarrow F(q,t)\}$

$\sigma_3 = \{v \leftarrow u\}$

$\sigma_4 = \{z \leftarrow F(q,t)\}$

$\sigma_5 = \{r \leftarrow F(u,u)\}$

$\sigma_6 = \{u \leftarrow G(F(q,t))\}$

$\sigma_7 = \{q \leftarrow s\}$

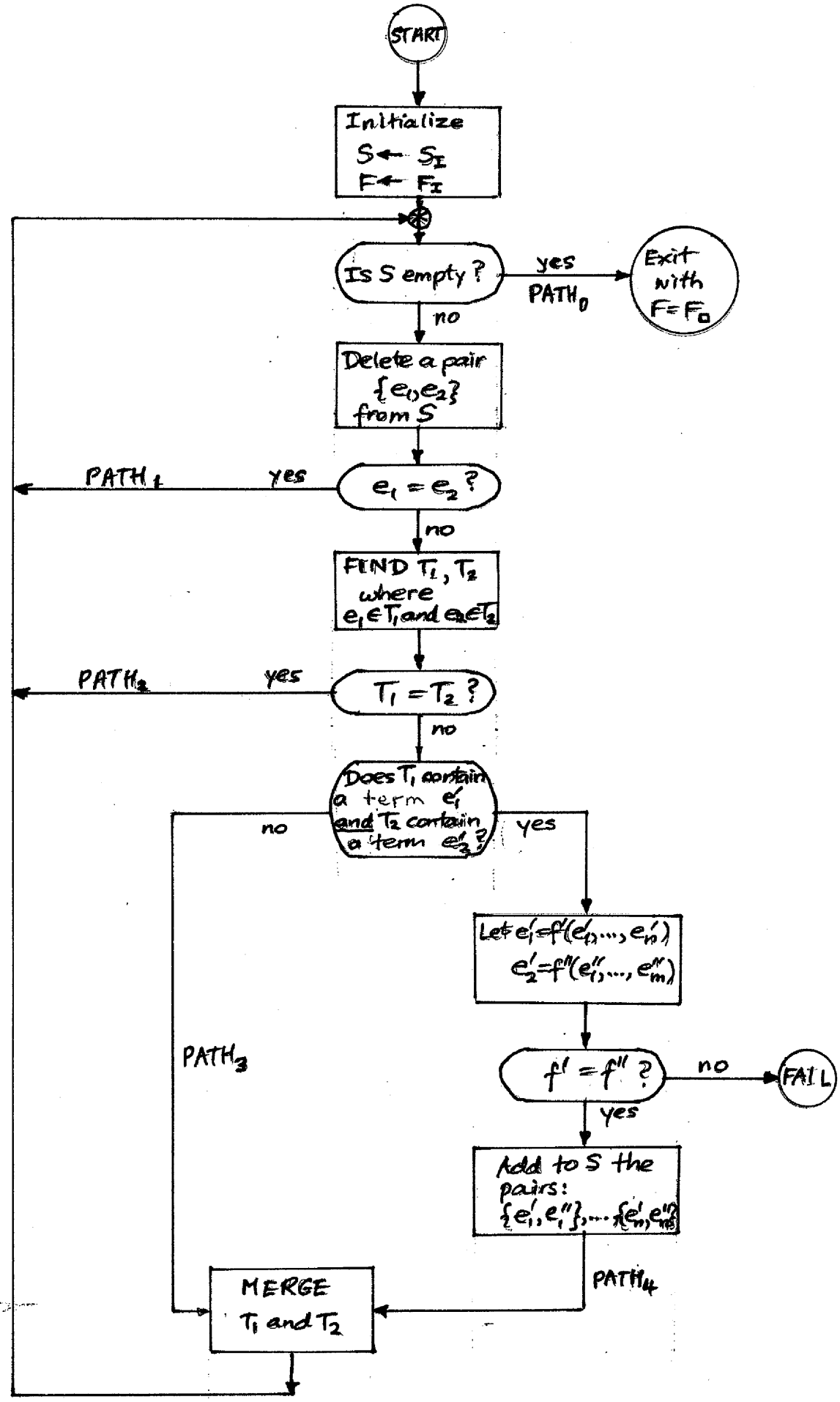
$\sigma_8 = \{t \leftarrow s\}$.

4. ALGORITHMS

In this section we present the flowchart for the transformational stage of our algorithm for unifying a set S_I of pairs of expressions. Since the flowchart is abstract, in that we do not specify the data structures used nor do we describe how the operations are performed, we also give a structured program, TRANSFORM. This uses several subroutines: DECOMPOSE, which unifies two terms by extracting their arguments, FIND which determines the set an expression belongs to and MERGE which constructs the union of two sets.

We will not present a topological sorting algorithm since this is readily obtained from the literature [4], however we give programs to communicate with it. CONSTRUCT DIRECTED GRAPH constructs from the final forest F_0, output by the transformational stage, a directed graph which is then used as input by the sorting algorithm. The output sorted graph is then used to obtain the most general unifier by OUTPUT UNIFIER: if $v_1,\ldots,v_n$ is the output of the topological sorting algorithm then OUTPUT UNIFIER $(v_1),\ldots,$OUTPUT UNIFIER (v_n) gives the required unifier.

An example using the transformational stage is given in Figure 4.1 where reference is made to the flowchart.



Flowchart for the transformational stage of the algorithms.

Example illustrating the transformational stage of the algorithm for $S_I = \{\{e1,e2\}\}$ where

$e1 = P(x,G(F(x,w)),v,F(F(u,u),t),x)$

and $e2 = P(F(G(y),G(z)),u,G(F(r,s)),y,F(u,v))$.

The following tables indicate the status of S and F for each cycle of the algorithm.

C	D	F
0	1	{e1}
0	2	{x}
0	3	{GFxw}
0	9	{Fxw}
0	12	{w}
0	4	{v}
0	5	{FFuut}
0	11	{Fuu}
0	3	{u}
0	12	{t}
0	1	{e2}
0	2	{FGyGz}
0	7	{Gy}
0	5	{y}
0	8	{Gz}
0	10	{z}
0	4	{GFrs}
0	10	{Frs}
0	16	{r}
0	17	{s}
0	6	{Fuv}

C	D	F
1		{e1,e2}
2	6	{x,FGyGz}
3	7	{GFxw,u}
4	8	{v,GFrs}
5	9	{FFuut,y}
6	11	{x,FGyGz,Fuv}
7	14	{GFxw,u,Gy}
8	14	{v,GFrs,Gz}
9	15	{FFuut,y,Fxw}
10	15	{Frs,z}
11	16	{Fuu,x,FGyGz,Fuv}
12	17	{t,w}
14		{GFxw,u,Gy,v,GFrs,Gz}
15		{FFuut,y,Fxw,Frs,z}
16		{Fuu,x,FGyGz,Fuv,r}
17		{t,w,s}

C = 0 gives F = F_I

D = " " gives F = F_0

C	D	P	S
0	1	4	{e1,e2}
1	2	3	{x,FGyGz}
1	3	3	{GFxw,u}
1	4	3	{v,GFrs}
1	5	3	{FFuut,y}
1	6	4	{x,Fuv}
6	7	4	{Gy,u}
6	8	4	{Gz,v}
7	9	4	{Fxw,y}
8	10	3	{Frs,z}
9	11	4	{Fuu,x}
9	12	3	{t,w}
11	13	2	{u,Gy}
11	14	4	{u,Gz}
14	15	4	{Fxw,Frs}
15	16	3	{Fuu,r}
15	17	3	{t,s}
	18	0	

C = 0 gives S = S_I

P = 0 gives S = ϕ

"C" indicates at what cycle a set was created; "D" indicates at what cycle a set was deleted. "P" refers to the particular path in the flowchart taken during a cycle. Underlined expressions are the specified terms in sets of F.

The left table gives F_I, the middle table gives the rest of F during the algorithm and the right table indicates S.

Figure 4.1

```
TRANSFORM:
begin
   INITIALIZE STACK and FOREST;
   repeat until STACK is empty:
   begin
      POP a pair of expressions {e1,e2} off STACK;
      if e1 ≠ e2 (by virtue of different pointers)
      then begin
             root1 ← FIND(e1); root2 ← FIND(e2);
             if root1 ≠ root2
             then begin
                    if TERM [root1] ≠ null and
                       TERM [root2] ≠ null
                    then DECOMPOSE (TERM [root1], TERM [root2]);
                    MERGE (root1,root2);
                  end;
          end;
   end;
end.

DECOMPOSE (expr1, expr2):
begin
   if SYMBOL [expr1] ≠ SYMBOL [expr2]
   then EXIT with FAILURE
   else for each argument pair arg1, arg2
            found from ARGLIST [expr1], ARGLIST [expr2] do
            PUSH arg1 and arg2 onto STACK;
end.
```

```
FIND (vertex):
begin
   comment use the top of STACK as a temporary LIST;
   v ← vertex;
   repeat until PARENT [v] = null:
   begin
      add v to LIST;
      v ← PARENT [v];
   end;
  return (v) since it is now the root;
  comment collapse the path directly onto the root;
  for each w on LIST do PARENT [w] ← v;
end.
```

```
MERGE (root1, root2):
begin
   assume that COUNT [root1] ≤ COUNT[root2]
   otherwise swap root1 and root2 in :
   begin
      light ← root1; heavy ← root2;
      PARENT [light] ← heavy;
      COUNT [heavy] ← COUNT [heavy] + COUNT [light];
      comment if necessary, update the term attached to the new root;
              if TERM [light] ≠ null
              then TERM [heavy] ← TERM [light];
   end;
end.
```

```
CONSTRUCT DIRECTED GRAPH:
begin
   for all variables, v in F0 do
       if PARENT [v] ≠ null, that is, v is not a root
       then begin
               root ← FIND(v);
               add the edge: v → root;
            end;
   for all roots, r in F0 do
       if TERM[r] ≠ null
       then RECORD(r, TERM[r]);
end.
```

```
RECORD (expr, expr'):
begin
   comment for each variable var' occurring in expr'
           add the edge:  expr → var';
   if VARIABLE [expr']
   then add the edge : expr → expr'
   else for each argument of expr' do
        RECORD (expr, argument);
end.
```

```
OUTPUT UNIFIER (variable):
begin
   if VARIABLE [variable]
   then comment CONSTRUCT DIRECTED GRAPH has already
                made all variables point directly to the root;
        if PARENT [variable] ≠ null
        then OUTPUT the substitution, {variable ← PARENT [variable]}
        else if TERM [variable] ≠ null
             then OUTPUT the substitution, {variable ← TERM [variable]};
end.
```

5. CORRECTNESS PROOF

We will prove that if our algorithm comprising the transformational stage and the sorting stage halts it does so correctly. That is, if failure of unification is reported then indeed the input set S_I is not unifiable and if a substitution is output then indeed this is a most general unifier (mgu) for S_I. That our algorithm halts will be evident from the timing analysis of the next section.

We will prove that if failure is reported during the transformational stage then S_I is not unifiable. If this stage exists successfully we will prove that any mgu for F_0 is also a mgu for S_I. During the sorting stage, if a circuit is detected in the associated directed graph then we will prove that F_0 is not unifiable and hence S_I is not unifiable. If a substitution is output we will prove that it is a mgu for F_0, and hence also for S_I.

In the transformational stage we will show:

$$\forall \sigma \ (\sigma \text{ is a mgu of } S_I \iff \sigma \text{ is a mgu of } F_0) \qquad \ldots (1)$$

This is a consequence of the easier condition:

$$\forall \sigma \ (\sigma \text{ unifies } S_I \iff \sigma \text{ unifies } F_0) \qquad \ldots (2)$$

To prove this we consider the following invariant property of the transformational flowchart:

$$\forall \sigma \ (\sigma \text{ unifies } S_I \iff \sigma \text{ unifies } S \ \& \ \sigma \text{ unifies } F) \qquad \ldots (3)$$

Initially (3) is true since S equals S_I and since each set of F is initially a singleton. If we can prove that (3) remains invariant then if the exit is successful with S empty and F equal to F_0 then (3) becomes (2).

To prove that (3) remains invariant, we prove that if (3) holds at the point of the flowchart labelled "*" then the next time this point is reached (3) remains true. It suffices to prove:

$\forall\sigma$ (σ unifies S' & σ unifies F' $\leftrightarrow$ σ unifies S" & σ unifies F") ... (4)

where S' and F' are the old values of S and F at "*" and S" and F" are the new values of S and F when "*" is next reached. Simple properties of sets, expressions and substitutions are used in the proof which has already been informally justified in section 3. We must examine each possible path in the flowchart which starts and finishes at "*".

If any path ends in failure then S_I is not unifiable because:

$f' \neq f'' \rightarrow \sigma$ does not unify $\{f'(e'_1,\ldots,e'_n), f''(e''_1,\ldots,e''_m)\}$ and X is not unifiable & $X \subseteq Y \rightarrow Y$ is not unifiable.

Consider $PATH_1$ of the flowchart. Here $S' = S'' + \{e_1,e_2\}$ (that is, $S \cup \{\{e_1,e_2\}\}$) and $F' = F''$. (4) is trivially true since any σ unifies $\{e_1,e_2\}$ when $e_1 = e_2$.

Consider $PATH_2$. Here $S' = S'' + \{e_1,e_2\}$. Also where $e_1 \in T_1$ and $e_2 \in T_2$, $T_1 = T_2 \in F' = F''$ hence (4) is true since:

σ unifies $\{e_1,e_2\}$ & σ unifies T_1 iff σ unifies T_1.

For $PATH_3$ $S' = S'' + \{e_1,e_2\}$. Let $F' = F \cup \{T_1,T_2\}$ then $F'' = F \cup \{T_1 \cup T_2\}$. (4) holds since:

σ unifies $\{e_1,e_2\}$ & σ unifies T_1 & σ unifies T_2 iff σ unifies $T_1 \cup T_2$.

Finally we consider $PATH_4$. Let $S' = S + \{e_1,e_2\}$ then $S'' = S \cup \{\{e'_1,e''_1\},\ldots,\{e'_n,e''_n\}\}$ where $e_1 = f'(e'_1,\ldots,e'_n)$ and $e_2 = f''(e''_1,\ldots,e''_m)$. Let $F' = F \cup \{T_1,T_2\}$ then $F'' = F \cup \{T_1 \cup T_2\}$. (4) is valid since:

σ unifies $\{e_1,e_2\}$ & σ unifies T_1 & σ unifies T_2

iff σ unifies $\{\{e_1',e_1''\},\ldots,\{e_n',e_n''\}\}$ & σ unifies $T_1 \cup T_2$.

We can also formally verify that the special properties of the sets of F are invariant: all the terms in each set of F begin with the same function-symbol and each subexpression of S_I belongs to exactly one set of F. These properties are initially true; the first property is ensured due to the fact that on $PATH_4$, f' = f" and the second property is preserved after each merge. These facts are important when we consider the sorting stage.

We must now show that the sorting stage is correct. We will first show that if a circuit exists in the associated directed graph then F_0 is not unifiable. By the construction of this graph, a circuit implies a sequence of trees in the forest F_0: $T_0,T_1,\ldots,T_n$, a sequence of roots of these trees: $r_0,r_1,\ldots,r_n$ and a sequence of vertices: $v_0,v_1,\ldots,v_n$ with the following properties:

v_i is a vertex (possibly the root) in tree T_i.

The set corresponding to T_i contains a term e_i in which the variable associated with vertex v_{i+1} occurs. (Addition is modulo n).

Denote these terms by $e_i[v_{i+1}]$ where we identify vertices with variables. Then if there is a unifying substitution, σ we must have:

σ unifies F_0

$\rightarrow$ σ unifies each tree in F_0

$\rightarrow$ in particular, σ unifies $T_0,T_1,\ldots,T_n$

$\rightarrow$ $\sigma(v_i) = \sigma(r_i) = \sigma(e_i[v_{i+1}])$ for $i = 0,1,\ldots,n$

$\rightarrow$ length $[\sigma(v_i)]$ = length $[\sigma(e_i[v_{i+1}])]$

> length $[\sigma(v_{i+1})]$

$\rightarrow$ length $[\sigma(v_0)]$ > ... > length $[\sigma(v_n)]$ > length $[\sigma(v_0)]$.

■

The contradiction implies that F_0 is not unifiable.

We now prove that the output of the sorting stage is a mgu of F_0. We do this by constructing a set S_2 containing two expressions from F_0 and applying Robinson's original unification algorithm. We show that any mgu of S_2 is also a mgu of F_0, and that the substitution σ_0 given by the sorting stage is indeed a mgu of S_2 since it is output by Robinson's algorithm which he proved correct. Therefore σ_0 is also a mgu of F_0.

The constructed set S_2 is $\{P(v_1,\ldots,v_n),P(e_1,\ldots,e_n)\}$ where P is a new n-ary function-symbol and $\sigma_0 = \{v_n \leftarrow e_n\}\ldots\{v_1 \leftarrow e_1\}$ is the substitution induced by the sorting stage. To show that:

$$\forall\ \sigma\ \ (\sigma \text{ is a mgu of } F_0 \iff \sigma \text{ is a mgu of } S_2)$$

we need only prove:

$$\forall\ \sigma\ \ (\sigma \text{ unifies } F_0 \iff \sigma \text{ unifies } S_2).$$

Now the sorting stage outputs all vertices of the forest corresponding to variables and by the construction of σ_0 the last statement is true; this would be the case for any permutation of the variables.

It remains to prove that the most general unifier output by Robinson's algorithm when applied to S_2 is exactly σ_0. Let $v_1,\ldots,v_n$ be Robinson's "lexical ordering" of the variables then when we scan S_2 in parallel and come to a disagreement set $\{v_i,e_i\}$ if v_i is a variable and e_i a term, then v_i does not occur in e_i otherwise a circuit would have been detected. We apply the substitution $\{v_i \leftarrow e_i\}$, however this does not effect any expressions to the right since all the v_i's are distinct and have been topologically ordered. If e_i is a variable then v_i must precede e_i in lexical order and

hence the substitution to be applied is still $\{v_i \leftarrow e_i\}$. By repeating this scanning process we see that the mgu obtained is identical with σ_0.

6. TIMING ANALYSIS

In this section we will show that the total amount of time taken by the transformational stage is bounded above by O(nG(n)) and similarly for the time to construct the directed graph. The time taken during the sorting stage is O(n). Consequently the total time has an upper bound of O(nG(n)).

Throughout this section n is the length of the input set, S_I, where we extend the measure, length, to sets of pairs of expressions. We allow our sets in S to contain repetitions.

$$\text{length}[S] = \sum_{\{e_1,e_2\} \in S} \text{length}[e_1] + \text{length}[e_2].$$

We now define the function G and show that it is "practically" constant. First we introduce an extremely fast growing function, H, by the following definitions.

$$H(i) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} \qquad \text{where there are } i \geq 1 \text{ two's.}$$

$$H(0) = 1$$

For example $H(4) = 65536$ and $H(5) = 2^{65536}$.

G is the "inverse" of H, that is, G(n) is defined as the least integer k for which $H(k) \geq n$. G grows extremely slowly: $G(n) \leq 5$ for all "practical" values of n, that is, for all $n \leq 2^{65536}$.

We will first analyze the transformational stage. By neglecting the cost of FIND and MERGE instructions we will show that the number of steps required is linear relative to the length of the input, n. Consequent-

ly there can only be a linear number of FIND and MERGE instructions. The following theorem from [1] tells us the total cost of these instructions, in which the cost of a MERGE is constant and the cost of FIND(v) is proportional to the number of vertices on the path from the vertex labelled v to the root of the tree containing this vertex.

Theorem Let c be any constant. Then there exists another constant c' depending on c such that the algorithm, which uses balancing and collapsing heuristics, will execute a sequence of c·n MERGE and FIND instructions on n elements in at most c'nG(n) steps.

Hence the total number of steps taken during the transformational stage to transform S_I, of length n, into F_0 is at most O(n·G(n)). Further, the length of F_0 is no greater than that of S_I.

We will now examine the operations in our algorithm, referrring to the flowchart of section 4. From examining the detailed data structures and programs given, if we ignore the cost of FIND and MERGE instructions, each basic operation of the flowchart, except one, requires only a constant amount of time. For example, to determine if a set contains a term requires merely the inspection of the TERM field of the root of the tree which represents the set. The exceptional case is "add to S all the pairs of arguments" since there are as many pairs as there are arguments of the term. However we can easily "absorb" this additional cost by effectively counting the edges rather than the vertices in the tree representation of an expression.

Proceeding formally, let TIME[S;F] be the additional time taken to process the sets S and F when the algorithm reaches the point labelled "*" in the flowchart. We now set up a system of recursive equations involving

TIME, by considering all possible paths.

If the exit is taken after finding S empty we have:

$$TIME[\phi;F_0] = c_1$$

where c_1 is the constant time taken to determine if S is empty.

If $PATH_1$ is taken then

$$TIME[S + \{e_1,e_2\};F] = TIME[S;F] + c_2.$$

If $PATH_2$ is taken, we have similarly

$$TIME[S + \{e_1,e_2\};F] = TIME[S;F] + c_3$$

where we initially neglect the cost of a FIND.

If $PATH_3$ is taken then

$$TIME[S + \{e_1,e_2\};\ F \cup \{T_1,T_2\}] = TIME[S;F \cup \{T_1 \cup T_2\}] + c_4$$

where the cost of a MERGE is ignored.

If $PATH_4$ is taken then

$$TIME[S + \{e_1,e_2\};\ F \cup \{T_1,T_2\}]$$

$$= \begin{cases} TIME[S \cup \{\{e_1',e_1''\},\ldots,\{e_m',e_m''\}\};\ F \cup \{T_1 \cup T_2\}] + c_5m + c_6 \\ \qquad\qquad \text{if } f' = f'' \\ c_7 \qquad\qquad \text{if } f' \neq f'' \end{cases}$$

where c_5m is the time taken to add to S the m pairs of arguments and c_6 is some constant time, just as is c_1,c_2,c_3 and c_4.

To simplify these equations we absorb the c_5m term into the TIME measure by the transformation:

$$TIME'[S;F] = TIME[S;F] + c_5.pairs[S]$$

where pairs[S] is the number of pairs in the set S. Also we will replace each equality by "≤" and each c_i by the maximum, c_{max}, of all such c_i's. Finally the transformation:

$$T[S;F] = TIME'[S;F]/c_{max}$$

transforms each c_i into unity. Without loss of generality we can assume that $c_1 = c_7 = 0$. Since we have

$$TIME \leq TIME' = c_{max} \cdot T$$

once we show that T is linear, it follows that TIME is also.

We now prove that T[S;F] is linear relative to the lengths of S and F. In fact we will prove that

$$T[S;F] \leq pairs[S] + arity[F]$$

where the measure, arity, is the sum of all "arities" of each set in F. We will formally define arity for expressions.

$$arity[f(e_1,\ldots,e_m)] = m,$$

that is, if f is a function-symbol then $m \geq 0$ gives the number of arguments, otherwise when f is a variable then its arity is zero. Extending arity to sets of expressions in which all terms, if any, begin with the same function-symbol we have:

$$arity[T] = \begin{cases} arity[e] & \text{if T contains a term, e} \\ 0 & \text{if T contains only variables.} \end{cases}$$

Finally we extend arity to collections of such sets:

$$arity[F] = \sum_{T \in F} arity[T].$$

We now prove our time estimate by induction on pairs[S] + arity[F].

Our set of recursive equations is, after the transformations:

$PATH_0$: $T[\phi;F_0] = 0$

$PATH_{1,2}$: $T[S + \{e_1,e_2\};F] \leq T[S;F] + 1$

$PATH_3$: $T[S + \{e_1,e_2\}; F \cup \{T_1,T_2\}] \leq T[S;F \cup \{T_1 \cup T_2\}] + 1$

$PATH_4$: $T[S + \{e_1,e_2\}; F \cup \{T_1,T_2\}]$

$$\begin{cases} \leq T[S \cup \{\{e_1',e_1''\},\ldots,\{e_m',e_m''\}\}; F \cup \{T_1 \cup T_2\}] + 1 \\ \qquad \text{if } f' = f'' \\ = 0 \qquad \text{if } f' \neq f'' \end{cases}$$

For each equation we will now apply our induction.

$PATH_0$: $T[\phi;F_0] = 0 \leq pairs[\phi] + arity[F_0]$

is trivially true for our basis of induction.

$PATH_{1,2}$: $T[S + \{e_1,e_2\};F] \leq T[S;F] + 1$

$\leq pairs[S] + arity[F] + 1$

(by induction hypothesis)

$= pairs[S + \{e_1,e_2\}] + arity[F]$

$PATH_3$: $T[S + \{e_1,e_2\}; F \cup \{T_1,T_2\}]$

$\leq T[S;F \cup \{T_1 \cup T_2\}] + 1$

$\leq pairs[S] + arity[F] + arity[T_1 \cup T_2] + 1$

(by induction hypothesis)

$= pairs[S + \{e_1,e_2\}] + arity[F] + arity[T_1] + arity[T_2]$

$= pairs[S + \{e_1,e_2\}] + arity[F \cup \{T_1,T_2\}]$

Here $arity[T_1 \cup T_2] = arity[T_1] + arity[T_2]$

since one of T_1, T_2 contains no terms.

$PATH_4$: The result is trivial when $f' \neq f''$, otherwise

$$T[S + \{e_1,e_2\};\ F \cup \{T_1,T_2\}]$$
$$\leq T[S \cup \{\{e_1',e_1''\},\ldots,\{e_m',e_m''\}\};\ F \cup \{T_1 \cup T_2\}] + 1$$
$$\leq pairs[S] + m + arity[F] + m + 1$$

(by induction hypothesis)

$$= pairs[S + \{e_1,e_2\}] + arity[F \cup \{T_1,T_2\}]$$

(since $arity[T_1] = arity[T_2] = m$).

Hence, in particular

$$T[S_I;F_I] \leq pairs[S_I] + arity[F_I].$$

This quantity is clearly proportional to the length of S_I. Therefore, ignoring FIND and MERGE instructions, the time taken during the transformational stage of our algorithm, $TIME[S_I;F_I]$ is linear relative to n, the length of S_I.

The number of elements in the forest, F is always the total number of subexpressions in S_I, which is also proportional to the length of S_I. Hence the additional time required to execute a sequence of $c \cdot n$ FIND and MERGE instructions on the expressions in F is bounded above by $O(nG(n))$.

During the construction of the directed graph associated with F_0 we also execute FIND instructions; the total time is similarly at most $O(nG(n))$.

The topological sorting stage requires time $O(V+E)$ where V is the number of vertices and E is the number of edges in the directed graph. This is proved in [4]. Here, V is at most the number of subexpressions in F_I and E is proportional to the length of S_I; both are linear relative to n, the length of S_I.

Finally, the total time taken by our algorithm, which includes the transformational stage, construction of a directed graph and the topological sorting stage is at most $O(nG(n))$.

7. CONCLUSION

From a computational complexity viewpoint the unification algorithm presented is the best known, being bounded above by O(nG(n)) time. Since it uses the set manipulation algorithms of [1] the algorithm is not linear, as proved in that book. It has recently been proved [10] that G may be replaced by the "inverse" of Ackermann's notorious function (which generalizes: successor, addition, multiplication, exponentiation,...). The existence of a linear algorithm is possible and this could perhaps be achieved by choosing, during the transformational stage, a pair of expressions from S according to some heuristic, rather than processing them "on-line". Such an algorithm may eliminate an irritating feature of our algorithm: the simplest case of unification, in which only variables appear, requires more than linear time. (Of course, we could recognize this as a special case).

Practically, it would be difficult to choose between a hypothetical linear algorithm and our "practically" linear one or even with some earlier character manipulation algorithms in view of special hardware features of some computers. Theoretically we can find pathological examples which exhaust our algorithms and it would be interesting to examine the source of such curiosities.

It is easy to show that the space requirements are linear relative to the input length, however it would be interesting to see how little space is required. In particular, the additional space required by the topological sorting stage could perhaps be supplied by the PARENT, COUNT and TERM fields of the expressions.

In a future paper the complexity of first-order subsumption and second-order instantiation will be examined. A different direction of research would be to examine the contexts in which unification algorithms appear. For example, the failure of unification in one case may tell us immediately if some related case also fails.

REFERENCES

[1] Aho A.V., Hopcroft J.E. and Ullman J.D. (1974)
The Design and Analysis of Computer Algorithms, Addison-Wesley.

[2] Baxter L.D. (1973)
An efficient unification algorithm, Research Report CS-73-23, Department of Computer Science, University of Waterloo.

[3] Fischer M.J. (1972)
Efficiency of equivalence algorithms, in Complexity of Computer Computations, R.E. Miller and J.W. Thatcher (editors), Plenum Press.

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

[5] Munro I. (1971)
Efficient determination of the transitive closure of a directed graph, Information Processing Letters, 1, No. 2.

[6] Newell A., Shaw C. and Simon H. (1955)
The Logic Theory Machine: a complex information processing system, IRE Transactions on Information Theory IT-2 (1956), 61-79.

[7] Post E., (1940)
Absolutely unsolvable problems and relatively undecidable propositions, in The Undecidable

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

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

[10] Tarjan R.E. (1975)
Efficiency of a good but not linear set union algorithm, JACM 22 (1975) 215-225.

[11] Van Emden M.H. (1975)
Programming with resolution logic, Research Report CS-75-30, Department of Computer Science, University of Waterloo.

[12] Venturini Zilli M. (1975)
Complexity of the unification algorithm for first-order expressions, to appear in Calcolo.