LINEAR UNIFICATION

by

M. S. Paterson*
University of Warwick
Coventry, England CV4 7AL

and

M. N. Wegman
IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

Abstract

A unification algorithm is described which tests a set of expressions for unifiability and which requires time and space which are only linear in the size of the input.

1. Introduction

The problem of unification arises in work on theorem proving and has been extensively treated in the literature [1]. From the set U of finite terms over a set of function symbols (including constant) and a countable set of variable symbols $\{x_1,x_2,...\}$, there is given a finite set of pairs of terms, $\{<u_i,v_i>|i\in I\}$. The unification problem is to determine the existence of a substitution $\sigma=\{(x_j\to t_j) \mid t_j\in U, j>0\}$ such that $\sigma(u_i)=\sigma(v_i)$ for $i\in I$. The substitution is to be carried out simultaneously on all variable occurrences. Less formally and less generally, unification is accomplished by substituting expressions for variables in such a way as to make two expressions identical. Such a substitution σ is a *unifier* and for any unifiable set of pairs there is a *most general unifier*, (m.g.u.) which is unique to within a permutation of variables, and from which any other unifier can be produced by a further substitution.

Example 1

$$\{<F(x_1,x_2),F(G(x_2),G(x_3))>\}$$

is unifiable with m.g.u. $x_1\to G(G(x_3))$, $x_2\to G(x_3)$, $x_j\to x_j$ for $j>2$.

Example 2

$$\{<x_1, G(x_1)>\}$$

is not unifiable. With a somewhat different framework we might wish to regard this as unifiable with the infinite substitution $x_1\to G(G(...$, and we shall return to this briefly in a later section.

Example 3

$$\{<F(x_1,x_1),x_2>,<F(x_2,x_2),x_3>,...,$$
$$<F(x_{n-1},x_{n-1}),x_n>\}$$

is unifiable but the image of x_n in the m.g.u. is a term with 2^{n-1} occurrences of x_1. It is undesirable and fortunately unnecessary to represent substitutions in explicit form. In this case we may write just

$$x_n\to F(x_{n-1},x_{n-1}),\ x_{n-1}\to F(x_{n-2},x_{n-2}),...,x_2\to F(x_1,x_1)$$

where these replacements are to be carried out *sequentially* from the left, i.e. all occurrences of x_n are replaced and then all (old and new) occurrence of x_{n-1}, etc. In this form the m.g.u. for Example 1 is $x_1\to G(x_2)$, $x_2\to G(x_3)$. In our algorithm we shall always produce a substitution as such a sequence

$$x_{i_1}\to t_1,\ x_{i_2}\to t_2,...$$

where for each $j>0$, t_j contains no x_{i_k} with $k\leq j$.

2. Representation

Terms from U are naturally representable as directed trees with nodes labelled with function and variable symbols. However we have seen already that the size of trees we may wish to consider may grow exponentially as a result of a unification. We will therefore adopt initially a more economical representation using directed acyclic graphs (dags) in which common subexpres-

* The first author was partially supported by the CNR of Italy at the IAC, Rome in March, 1974; and by the NSF (grant MCS72-03752 A03), the ONR (N 00014-76-C-0330), and the IBM Corporation while at Stanford University in 1976.

sions are represented by a single subgraph. Example 1 and its unification are illustrated in Figure 1.

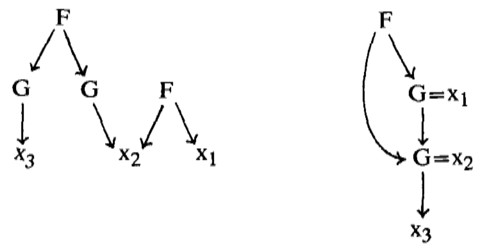


Figure 1

A node labeled with a k-ary function symbol ($k \geq 0$) has outdegree k and for $k > 1$ the outgoing arcs are distinguished so that we may refer to the first son, second son, ... A node labeled with a variable symbol has outdegree 0. We shall refer to these as *function* nodes (even when $k=0$ for constants) and *variable nodes*. Any node with indegree 0 is a *root*.

To simplify the discussion we shall consider problems in which only one pair of terms is to be unified. Any set of pairs can be combined so that corresponding terms occupy corresponding subterms in a pair of constructed terms, for example $\{<x_1,G(x_2)>, <F(x_1,x_2),x_3>\}$ would be replaced by $\{<F(x_1,F(x_1,x_2)), F(G(x_2),x_3)>\}$. This transformation involves only a linear increase in the length of input, the result is unifiable if and only if the original set is unifiable and moreover the m.g.u.'s are the same. Thus the input to our algorithm is to be a dag with a pair of distinguished nodes which are to be unified. If one of these nodes is an ancestor of the other then they are not unifiable and it should be evident that any ancestor nodes of the distinguished pair are irrelevant and could be deleted. Therefore we may as well assume that they are both roots of the dag.

Our aim is to be economical in our use of machine capabilities; for example, we shall not require features such as table look-up, hashing functions or random accessing. We shall use only operations which manipulate pointers and compare labels. Since we are considering substitutions which are applied uniformly to all occurrences of the same variable, it is a natural to assume that all pointers to a given variable are pointers to the same location.

3. Preliminary Results and Simple Algorithms

An equivalence relation on the nodes of a dag is *valid* if it has the following properties:

(i) if two function nodes are equivalent then their corresponding sons are equivalent in pairs,

(ii) each equivalence class is *homogeneous*, that is it does not contain two nodes with distinct function symbols,

(iii) the equivalence classes may be partially ordered by the partial order on the dag.

Properties (i) and (ii) ensure that if each equivalence class is coalesced to a single node then the resulting *reduced* graph may be consistently labeled with symbols and have arcs appropriate to the arities. Property (iii) is the condition that this reduced graph is acyclic.

Lemma 1. Nodes u,v are unifiable if and only if there is a valid equivalence relation with $u \equiv v$. In the affirmative case there is a unique minimal valid equivalence relation.

Proof. If $<u,v>$ is unifiable then let σ be a unifier. Define a relation Σ on nodes by $r \Sigma s$ if the images under σ of the terms represented by r and s are equal. Property (iii) holds, since the equivalence classes can be aptly labelled with the common term represented by its nodes after applying σ and the reduced graph gives just the sub-expression relation among a set of finite terms and so is acyclic. It is easily verified that $u \Sigma v$ and Σ is a valid equivalence relation.

On the other hand suppose Σ is a valid equivalence relation and $u \Sigma v$. From the (homogeneous) equivalence classes containing each of the variable nodes, a unifying substitution is immediately derivable.

Finally, if u and v are unifiable then consider the equivalence relation developed from $u \equiv v$ by combining equivalence classes only when required by condition (i). The resulting equivalence relation is contained in any valid equivalence relation for which $u \equiv v$ and so is itself valid. It is certainly uniquely minimal ■.

We shall initially describe algorithms from the point of view of an intelligent human with a blackboard and chalk. The following naive algorithm implements the procedure of the final part of the above proof. Some representation of an equivalence relation "≡" is maintained. Initially the relation is empty and then it is progressively augmented by "setting" certain pairs of elements to be equivalent, i.e. coalescing their two equivalence classes.

Algorithm A

```
Comment: tests the pair of nodes u,v for unifiability.
Set $u \equiv v$.
While (there is a pair of nodes r,s with $r \equiv s$ but with
   $r' \not\equiv s'$ for a pair of corresponding sons r′,s′), set $r' \equiv s'$.
Test the relation '≡' for conditions (ii) and (iii) of validity.
If '≡' valid, print 'UNIFIABLE' else print
   'UNUNIFIABLE' and halt.
```

End of Algorithm A.

In view of the crudity of the algorithms we shall not bother to prescribe the output of an m.g.u. The most obvious source of inefficiency in Algorithm A is that within some equivalence class of size k a number of pairs nonlinear in k may be checked or set equivalent in the 'while' loop, even if care is taken that no pair is set equivalent twice. Even with the best equivalence-handling algorithm known, a nonlinear amount of time would be required. Robinson has shown that similar techniques give an algorithm for obtaining an m.g.u. in almost linear time[2]. Setting sons equivalent when their fathers are equivalent, as is done in the "while" loop above, will be referred to as *propagation*. To achieve a linear algorithm we must propagate the equivalence relation in a carefully ordered way, taking one completed equivalence class at a time.

An equivalence class of dag nodes containing only roots will be called a *root class*. The crucial property of root classes is that no root class can be further extended by later propagation of the equivalence relation since propagation is only from fathers to sons. An important observation is the following.

Lemma 2. Any non-empty equivalence relation satisfying condition (iii) of validity has a root class.

Proof. Any maximal (i.e. nearest the roots) equivalence class in any partial order guaranteed by (iii) must be a root class ■.

Algorithm A may be improved as follows.

Algorithm B.

```
Set $u \equiv v$.
While there exists a root class $R = \{r_1, \ldots, r_k\}$ do
 begin
  If R is not homogenous, print 'UNUNIFIABLE'.
  If R contains a function node,
   let $r_1$ be a function node without loss of generality.
  For $i=2$ step 1 until k do
   If $r_i$ is a function node then
      For $j=1$ step 1 until (outdegree $(r_1)$) do
        Set $j^{th}son(r_i) \equiv j^{th}son(r_1)$
   else print '$r_i \to r_1$'.
  Delete nodes of R and their outgoing arcs.
 end
```

End of Algorithm B.

Note that in the equivalence class R we propagate just k-1 pairs, $<r_1 \equiv r_i>$ for $i=2,\ldots,k$. Since '≡' is treated as an equivalence relation this set is certainly adequate. If unification succeeds then in the final equivalence relation no further nodes are added to such a class R, because R is a root class. Thus R may indeed be correctly deleted after processing. The m.g.u. is output in the form of a sequence of substitution '$r \to s$' where in each case r is a variable node and s is another node. This is a convenient form for future processing and construction of dags.

Algorithm B checks complete equivalence classes for homogeneity before processing, but never needs to check the acyclicity condition, (iii). Suppose the minimal equivalence relation satisfying '$u \equiv v$' and conditions (i) and (ii) indeed had a cycle in its reduced graph. Then no class of the cycle could ever become a root class in the algorithm and so the algorithm could never terminate successfully.

We hope the reader needs no further persuasion as to the correctness of Algorithm B.

4. A Linear Algorithm

It remains to give an efficient procedure for finding a root class and to provide more details throughout.

```
Algorithm C
Comment: to test <u,v> for unifiability
   Create undirected edge (u,v)
   While there is a function node r, Finish (r).
   While there is a variable node r, Finish (r).
   Print "UNIFIED".

Procedure Finish (r)
  Create new pushdown stack with operations
       Push(*) and Pop.
  Push(r)
  While stack non-empty do
   Begin
    s:=Pop
    If Pointer (s) defined then begin
       If Pointer(s)≠r, Print "FAIL-LOOP" and halt
         end
      else
       Pointer (s):=r
    If r,s have different function symbols,
      print "FAIL-CLASH".
    While there is a father t of s, Finish (t).
    If s is variable node, print "s→r"
     else create undirected edges
       {(j^th son(r),j^th(s))|j=1,...,outdegree(r)}
    While there is an undirected edge (s,t),
       Push (t) and delete (s,t).
    If r≠s, delete node s and all dag arcs out of s.
   End.
   Delete node r and all dag arcs out of r.
End of Finish
End of Algorithm C.
```

An important feature of the procedure Finish is that it does not begin to look at the equivalence edges at a node s until all fathers of s have been deleted and s has become a root of the current graph. The recursive structure of Finish ensures that if work is begun on a non-root class this is suspended when a non-root is encountered and that each equivalence class which is completed is a root class. The total effect is therefore similar to Algorithm B in which a root class was "found" whenever needed. Notice that if Finish encounters a node which has already been met in a previous invocation then it is recognized by its having a pointer to a class representative different from the current one. The argument of Finish will always be a function node of the dag unless the dag has been reduced already to a set of isolated leaves.

To verify the linearity of Algorithm C given a reasonable data structure, we can partition the running time into segments of bounded size and assign these segments either to nodes of the dag, arcs of the dag or edges of the equivalence relation. It is easy to do this so that each is assigned only a bounded number of segments, noting only that iterations of the second 'while' loop of Finish should be assigned to the father t, rather than the son s. Now we observe that the total number of equivalence edges created is bounded above by the number of arcs of the dag, hence the total running time is linear in the "nodes + edges" of the dag and thus presumably linear in the length of the input representation. For a pointer machine of the kind we are considering, the space required cannot exceed the running time.

5. Data Structures

We will review here the basic capabilities required for a linear implementation of Algorithm C. It seems to be easier to mark nodes as "non-existent" instead of actually deleting them.

For the global structure we need, in order to

(I) find a new function node or variable node,
  a list of pointers to all nodes, with function nodes preceding variable nodes, though a careful traversal of the dag structure itself would suffice
(II) manage recursion and local pushdown stacks,
  a pushdown stack

For each node, s, of the dag we need, in order to

(i) determine the existence and symbol of s,
  a label which may take as its value 'DELETED', 'VARIABLE' or any function symbol,
(ii) determine if s is a root
  a list of pointers to fathers of s,
(iii) enumerate sons of s
  a list of pointers to sons
(iv) represent equivalence edges
  a list of edges (s,t) incident with s
(v) to indicate Pointer (s)
  a pointer.

The fathers-list of each node is used as follows. A pointer is moved along the list from "left" to "right" and whenever another father is sought the pointer is moved to the next father which is not marked 'DELETED'. Thus the total number of pointer moves made is just the number of fathers of the node in the original dag. This method obviates the need for backward pointers to a linked list. A similar procedure is used on the edge-lists.

6. Conclusion

Returning to Algorithm C one can see that the acyclicity condition is no longer an extra check at the end but is an integral part of the correctness and efficiency of the algorithm. Indeed we know of no linear algorithm for unification with infinite expressions. The current best algorithm maintains an "on-line" equivalence relation using the well-known UNION-FIND algorithm, which has a worst-case running time which is very slowly growing but non-linear [2,3]. This version of unification has applications to solving the equivalence problem for finite automata or to unification in a structure like the λ-calculus in which there is a fixed-point operator.

Though our concern for linearity is primarily of theoretical importance we believe that our algorithm is sufficiently simple and economical to implement, compared with others which have been proposed, that it may have a useful future in the practical domain.

Acknowledgements

We have had useful conversations with and comments on this work from Larry Carter, Marisa Venturini-Zilli, Bob Tarjan, Don Knuth, Paul Gilmore, and Alan Robinson

References

1. Robinson, J. A. 'Computational logic: the unification computation'. *Machine Intelligence 6*, 63-72.
2. Robinson, J. A. private communication, October 1975.
3. Tarjan, R. E. 'Efficiency of a good but not linear set union algorithm.' *JACM 22*, 215-225.

Appendix: a worked example.

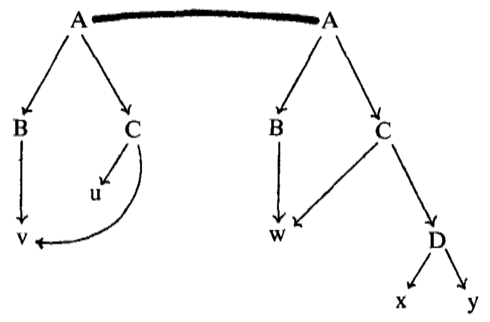
We will attempt to give some of the intuition behind the linear algorithm, by working an example, and highlighting the important steps. We will leave out certain operations, which are needed by the algorithm, because either in our example they are unnecessary or they obscure the example.

The question is, is

$$\{<A(B(v),C(u,v)), A(B(w),C(w,D(x,y))>\}$$

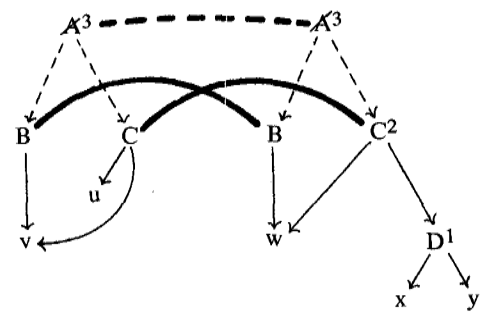
unifiable?

First we make a dag and add an undirected edge between the two roots. An undirected arc will be represented pictorially as a very wide line.

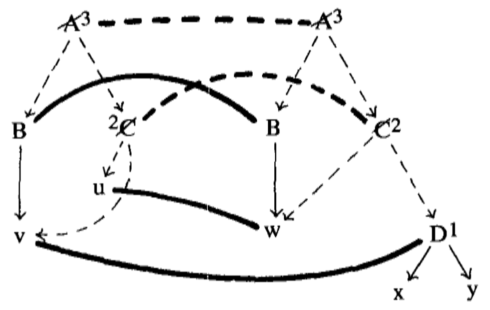


Suppose D(x,y) is the first function node we encounter. We will define its pointer as being to itself. We will represent this pictorially by adding a number 1 to label the node. We then explore its father C, by recursively invoking Finish. C's pointer is labelled with a 2 and we again recursively explore A, C's father, labelling A with a 3. (We leave it to the reader to understand which C node or A node we mean. We chose numbers for pointers because they are unique.) As there are no fathers of this A, we travel across the undirected arc to the other A and label it with a 3.Undirected arcs are drawn between corresponding sons of the A nodes, and the A nodes and associated arcs are deleted.We will pictorially denote a delete arc by a dashed line, and a deleted node by a slash through it.

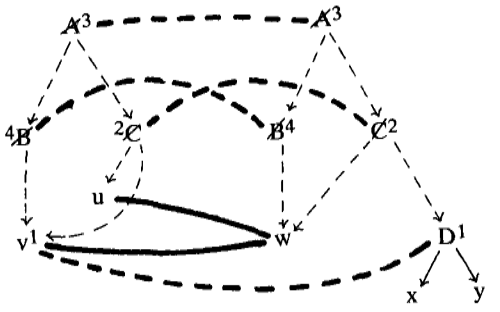
The picture now looks like this:



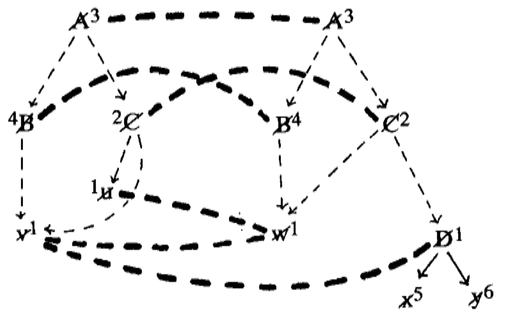
Since we are finished with the call Finish(A), we return to processing the equivalence class that will eventually be marked with a 2. We travel across to the other C node, draw arcs between corresponding sons, and delete the C nodes and their associated arcs.



We now return to D. We travel across to v, but before we delete v, we must visit its father. This step is necessary in the algorithm, otherwise we would miss the fact that D and v must be unified with u and w. The class with B in it is finished.



Finally, the first class we started with is finished, and the classes containing x and y are finished, and so are we.



A3
A3
4B
2C
B4
C2
1u
v1
w1
D1
x5
y6