Determining Logical Dependency in a Decision Procedure for Equality

Dean B. Krafft
Alan J. Demers

TR 81-458

April 18, 1981

Department of Computer Science
Cornell University
Ithaca, New York 14853

This work was supported in part by the National Science Foundation under Grants MCS80-03349 and MCS78-00953.

Determining Logical Dependency in a Decision Procedure for Equality*

Dean B. Krafft
Alan J. Demers
Cornell University

Abstract

Several existing program verification and automated proof systems make use of similar decision procedures for equality ([Krafft 1978], [Nelson and Oppen 1977] and [Downey, Sethi and Tarjan 1980]). The general method used by these algorithms has been named congruence closure. Given two expressions on uninterpreted function symbols, a congruence closure algorithm determines whether the expressions can be deduced to be equal from a set of previously asserted equalities. The existing congruence closure algorithms do not provide any indication of which previously asserted equalities were required in the deduction.

In this paper we describe an extension to one version of congruence closure. When two expressions are equal, the new algorithm can provide a certificate of their equality: a list of previously asserted equalities from which the queried equality can be deduced. This list is minimal, in the sense that if any equality is removed from the list, the queried equality can no longer be deduced. The running time to produce the certificate is almost-linear in the size of the list.

1. Introduction

In designing the AVID interactive program/proof development system [Krafft 1981], a major goal was to provide the user of the system with as much useful information about the proof as possible. One feature that seemed potentially valuable was allowing the display of logical dependency within the proof. This feature would allow the interactive display of dependency information about any proof statement, in the form of a minimal set of proof statements from which it logically followed. Alternatively, the system could highlight the proof statements that directly depended on another statement. Since the AVID system interacts directly with the user, a further requirement was that this information be determined promptly.

The programming logic used in AVID is that of PL/CV2 ([Constable and O'Donnell 1978], [Constable and Johnson 1979]). The implementation of a verifier for this logic is described in [Johnson 1981], and a modified version of this verifier is used in the AVID system. To reduce the

* This work was supported in part by the National Science Foundation under Grants MCS80-03349 and MCS78-00953.

Authors' address: Department of Computer Science, Upson Hall, Cornell University, Ithaca, NY 14853

programmer's effort and to simplify the presentation of proofs, the system automatically performs some simple deductions. For example, from the assertions "A => B" and "A", the assertion "B" can be deduced automatically, without explicit justification. As described in [Johnson 1981], the basis for almost all these automatic deductions is a decision procedure for the theory of equality. The general method used by this decision procedure has been named congruence closure.

Congruence closure algorithms, in off-line form, have been described in [Nelson and Oppen 1977] and [Downey, Sethi and Tarjan 1980]. The on-line algorithm used in the PL/CV2 system is based on an algorithm given in [Kozen 1977] as a solution to the uniform word problem for finitely presented algebras. This version of the congruence closure algorithm was first described in [Krafft 1978] and was proved correct in Chapter 6 of [Johnson 1981].

In this report we present an extended version of the congruence closure algorithm implementing the PL/CV2 equality decision procedure. The extension allows us to solve the logical dependency problem proposed above for deductions made by the decision procedure. The extension consists of a new procedure and some changes to the existing procedures and data structures. The new procedure, given two expressions that the congruence closure algorithm can show to be equal, produces a certificate of their equality. The certificate is a minimal list of existing equalities from which the expression can be deduced to be equal by applications of the axioms of equality.

As we will show later, adding the logical dependency extensions increases the running time of the standard congruence closure algorithm by only a small constant factor. The running time of the routine that produces certificates turns out to be almost-linear in the size of the certificate.

This report is organized as follows. In section 2, we present a simplified version of the logical dependency algorithm; it determines equalities on only simple variables instead of on full expressions. The equality decision procedure in this case is almost exactly McIlroy and Morris's UNION-FIND algorithm [Aho, Hopcroft and Ullman 1974]. In section 3, we review the decision procedure for equality presented in [Johnson 1981]. In section 4, we show how to integrate the algorithms of sections 2 and 3 to provide decision procedures for equality with certificates. In section 5, we outline proofs of correctness for the new algorithms. In section 6, we discuss their running time. Finally, in section 7 we present some conclusions and general observations.

2. A Simple Certification Algorithm

In this section we introduce the reader to the basic concepts. To this end, we present in section 2.1 a simplified version of our algorithm which solves the following problem: Execute a list of instructions consisting of equates of simple variables, queries on the equality of two simple variables, and requests for certification of equality between a pair of simple variables.

The sequence of pairs of simple variables that have been declared to be equal are referred to as EQS. Each equate instruction causes a pair to be added to the end of EQS. In response to a query on the equality of two simple variables, the algorithm returns true if their equality is deducible by means of repeated applications of reflexivity, symmetry, and/or transitivity on the pairs of variables in EQS, and false otherwise. A request for certification of equality for two simple variables returns a list of indices into EQS identifying a set of pairs that establish their equality. This list contains no duplicates.

Sequence EQS will not be used in the actual algorithms. It is simply used as an aid to presenting the problem being solved.

2.1. The Algorithm with Simple Certify Instructions

The algorithm used to determine equality for simple variables is based on the Union-Find algorithm [Aho, Hopcroft and Ullman 1974]. Every variable initially represents an equivalence class consisting only of itself. Each time two variables are declared equal, a Union operation is performed on their equivalence classes.

In the table below, we show the contents of a node in the Union-Find tree. We will refer to the fields by the names given here in our discussions of the algorithm.

field	contents
label	The variable name associated with this node.
reprlink	Either a pointer to a node, or nil (marking the root) such that for all nodes x and y, x and y are in the same equivalence class if and only if their reprlinks lead to the same root node.
reprcount	The number of nodes in this node's equivalence class (if this node is a representative). This value is used as the weight for the Union-Find algorithm.
witnesslink	Either a pointer to a node, or nil. If it is a pointer to a node, then the input contained a proclaim equating this node and the node pointed to.
witnesslabel	The index of the proclaimed equality that caused the witnesslink to be introduced. If witnesslink is nil, this value is undefined.
marked	false (used only within the Certify procedure).
next	The next entry in this hash bucket.
previous	The previous entry in this hash bucket. These fields are used to chain nodes together in a bucket used by the hash function of the Find algorithm.

The reader should be able to see that the Find algorithm of Figure 1, the Compare algorithm of Figure 2, and the Equate algorithm of Figure 3 (ignoring lines 12-21 of that algorithm) make use of the information described above to implement a weighted Union-Find without path compression.

```
Find(x):
¢ Return an entry for x in the data structure, creating it ¢
¢ if necessary.                                            ¢
begin
     h := hash(x)
  ;  Search bucket[h] for x
  ;  if  the search fails to find a node n
     then n:= a pointer to a newly allocated node
     ;    Put n in bucket[h], setting n.next and n.previous
     ;    n.label := x
     ;    n.reprcount := 1
     ;    n.reprlink, n.witnesslink := nil, nil
     fi
  ;  Find := n
end
```

Figure 1: Algorithm Find

```
Compare(xx, yy):
¢ Determine if variables xx and yy are equal ¢
begin
     x := Find(xx) ; y := Find(yy)
  ;  while x.reprlink ≠ nil do x := x.reprlink od
  ;  while y.reprlink ≠ nil do y := y.reprlink od
  ;  Compare := (x = y)
end
```

Figure 2: Algorithm Compare

To be able to certify equalities, the algorithm must keep a record of which variables were declared equal by input assertions. This is done by recording these asserted equalities as _witness edges_ between nodes of the Union-Find graph. The field "witnesslink" contains a pointer to represent this edge. Thus, there are either zero or one witness edges for a node. Each witness edge is labeled with the index in EQS of the input assertion of equality that caused its creation. This label is stored in the "witnesslabel" field of the node. The reader should observe that witness edges will coincide with Union-Find edges only when the nodes being declared equal are the representatives of their respective equivalence classes.

To certify that two variables are equal, we need only find a path through the witness edges from one variable to the other; the labels on the edges of the path form the desired certificate.

The problem is to find this path as quickly as possible. To do so, we established the following invariant:

```
Equate(xx, yy):
¢ Assert the equality of variables xx and yy ¢
begin
     Equate_index := Equate_index + 1
     x := Find(xx) ; y := Find(yy)
;    while x.reprlink ≠ nil do x := x.reprlink od
;    while y.reprlink ≠ nil do y := y.reprlink od
;    if   x ≠ y
     then ¢ Let x be the name of the new representative ¢
          ;    if y.reprcount > x.reprcount
                    then x,y := y,x fi
     ;    y.reprlink := x ; x.reprcount := x.reprcount + y.reprcount
          ¢ Reverse the witness path (yy, ..., y)                     ¢
          ;    w := yy
          ;    previouslabel := w.witnesslabel ; v := w.witnesslink
          ;    while v ≠ nil do
               ;    previouslabel, v.witnesslabel := v.witnesslabel,
                            previouslabel
               ;    v, v.witnesslink, w := v.witnesslink, w, v
               od
          ;    yy.witnesslabel := Equate_index ; yy.witnesslink := xx
     fi
end
```

Figure 3: Algorithm Equate

Invariant: The directed witness path from any member of an equivalence class ends in the (Union-Find) representative of that equivalence class.

With this invariant, finding a witness path between two equivalent nodes is straightforward. From each node, trace the witness path to the representative. The symmetric difference of these paths will give a path along witness edges between the nodes. This is the algorithm used by the Certify routine of Figure 4.

We argue that the nodes and witness edges form a directed graph in which each node has out-degree zero or one. By the invariant, this graph is acyclic (since every path ends); thus the graph is a forest. Further, the roots of this forest are exactly the Union-Find representatives (since every path ends at a Union-Find representative); thus, each witness equivalence class is a single tree in the forest. Observe that the height of this tree may be the number of nodes in the equivalence class. The height of the Union-Find tree, on the other hand, will be bounded by the logarithm of the number of nodes in the equivalence class.

In designing the complete algorithm we are faced with one more major problem: How can we perform Equate (Union) operations while maintaining the invariant necessary for Certification? The solution is as follows: To Equate two variables, choose the variable in the smaller equivalence class, reverse the edges on the witness path from that variable to its representative, and add the new witness edge from that variable to the other variable. Then do a normal (weighted) Union on the two equivalence classes.

Let us examine this solution, making use of the example shown in Figures 5 and 6. In this example, **b** and **d** are the initial representatives of the two equivalence classes. An Equate operation is being done on **a** and **e**.

The first step is to choose the variable in the smaller equivalence class. That variable is **e**. After choosing a variable, the next step is to reverse the edges on the witness path from this variable (**e**) to its representative (**d**). By our invariant, at the start of the operation the witness path out of every node in the equivalence class leads eventually to the representative (**d**). After reversing the edges, there is a path from the representative (**d**) to the variable (**e**). Therefore, there is now a path from every node in the equivalence class to that variable.

The next step is to add a witness edge from this variable (**e**) to the variable (**a**) we are Equating with it. Since by the invariant there was already a witness path from **a** to its representative, **b**, there is now a witness path from every node in the smaller equivalence class (**d**, **e**, **f**) to **b**.

The final step is to take the Union of the two equivalence classes. This will make **b** the representative of the combined equivalence class, and the invariant will have been reestablished.

Returning to the problem of producing certificates, consider the data structure of Figure 6. To certify that **f=e**, the algorithm must calculate the symmetric difference of the witness path from **f** to the representative (**b**) and the path from **e** to **b**. If the algorithm fully traced these paths, it would follow two edges that could be completely ignored (**e-a**, and **a-b**) twice. In the Certify algorithm shown in Figure 4 the symmetric difference of the paths is calculated while avoiding this extra effort. This is done by stepping along the witness paths from the two nodes to the representative in parallel, using field "marked" to mark as we go. As soon as the marked paths intersect (at the nodes' lowest common ancestor in the witness tree), a simple path between the nodes will have been found. This technique serves to bound the amount of work done by Certify to almost-linear in the number of edges in shortest witness path between nodes.

2.2. Solving a Slightly Harder Problem

In this section we extend the algorithm of the previous sectin to allow a single instruction to request certificates of equality for an arbitrary number of pairs of simple variables. Again, this is a list of indices into EQS. Avoiding duplicates makes this a more difficult problem.

```
Certify(xx, yy):
¢  Assuming that xx and yy are equal, output a list of previously ¢
¢  asserted equalities to certify it.                             ¢
begin
     x := Find(xx) ; y := Find(yy)
;    x1 := x ; y1 := y
     ¢  Find the lowest common ancestor (lca) of x and y in the   ¢
     ¢  witness tree                                              ¢
     ;    while ¬ x1.marked and ¬ y1.marked and x ≠ y
          do
               if   x1.witnesslink ≠ nil
               then x1.marked := true ; x1 := x1.witnesslink
               fi
          ;    if   y1.witnesslink ≠ nil
               then y1.marked := true ; y1 := y1.witnesslink
               fi
          od
     ;    if   x1.marked or x1 = y1
          then lca := x1 ; overshoot := y1
          else lca := y1 ; overshoot := x1
          fi
     ¢  Clear any marks above the lowest common ancestor     ¢
     ;    t := lca
     ;    while t ≠ overshoot
          do   t.marked := false ; t := t.witnesslink od
     for t in {x, y}
     do   while t ≠ lca
          do   Output t.witnesslabel ; t.marked := false
          ;    t := t.witnesslink
          od
     od
end
```

Figure 4: Algorithm Certify

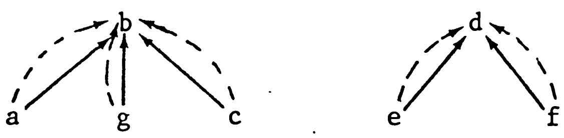

Figure 5: Data structure after a=b, b=c, d=e, d=f, g=b
Dashed lines indicate witness links, solid lines indicate representative links

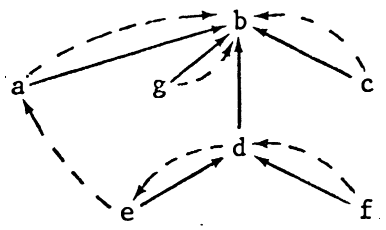

Figure 6: Data structure from Figure 5 after a=e

field	contents
witnessrepr	Within the MultiCertify routine, this is either a pointer to a node or nil such that for all nodes x and y, x and y are in the same witness equivalence class if and only if their witness paths end in the same root node.
witnesscount	If this node is a witness representative, this is the count of the number of nodes in its witness equivalence class. (Used only within the MultiCertify routine.)
realroot	Within the MultiCertify routine, if this node is the representative of a witness equivalence class, this is a pointer to the "real" root of that equivalence class. Initialized to point to the node itself. (See text for a fuller explanation.)

Figure 7: The additional node fields for a slightly harder problem

In determining logical dependency in the full congruence closure algorithm we need to deal with the problems posed by the presence of operands. We will reserve a complete discussion of these problems for section 4. For the moment, note that certifying the equality of two expressions involves: (1) observing that the outermost operators of the two expressions are equal, and (2) certifying that their operands are pairwise equal. Clearly, finding a minimal certificate for the one queried equality is equivalent to finding a minimal set of proclaimed equalities from which all the operand equalities can be inferred. In other words, we will want to consider the operand pairs collectively and produce one minimal certificate for all of them.

We now present a change to the Certify algorithm to do this. In Figure 7, we describe the new fields that must be added to the graph nodes to solve this problem.

The new Certify procedure changes the graph. In this and all following sections, we will assume that at the end of a series of Certify instructions, all these changes are undone in some fashion. We will simply note here that this can be done in time at most double that of the simple destructive algorithm.

To solve the new problem, we must avoid duplicating emitted equalities. At the same time, we must keep the running time of the algorithm as close as possible to O("number of emitted equalities"). Our solution is as follows: Form equivalence classes of all variables that can be shown to be equivalent by emitted equalities. These equivalence classes will be referred to as witness equivalence classes. The equality of two variables in the same witness equivalence class can be deduced from previously emitted certifying equalities. If two variables are not in the same witness equivalence class, then, as in the simple Certify algorithm, trace the shortest witness path between them. This time, however, emit only witness edges connecting previously disjoint witness equivalence classes, forming the union of these classes as the labels on the witness edges are emitted.

A full specification of this algorithm (MultiCertify) is shown in Figure 8. To implement the equivalence classes, we again make use of the Union-Find algorithm. In this case, the version used is the weighted, path-compressing, Union-Find, as presented in [Aho, Hopcroft and Ullman 1974]. The functions that implement the Union-Find operation, PathFind and PathUnion, are the standard algorithms, and are therefore not explicitly listed. PathFind is a function that returns the root node of the witness equivalence class for a node. PathUnion forms the union of two witness equivalence classes, returning the root node in the new class. These routines use the "witnessrepr" field for representative edges, and the "witnesscount" field for the weights on the nodes. Since the weighted, path-compressing, Union-Find is being used, the running time of the algorithm will be $O(N\ \alpha(N))$, where α is Tarjan's α function [Tarjan 1975]. For the MultiCertify algorithm, N will be the number of emitted equalities.

The MultiCertify operation makes use of one more special field. Each witness equivalence class must have a witness link exiting from it going toward the representative of the equality equivalence class. The real root of a witness equivalence class is the node that contains this witness link. This may not be the same node as that chosen by PathUnion as the Union-Find representative for the witness equivalence class. For the purpose of tracing the witness path, however, the real root is the node that must be taken as the one representing the equivalence class, since it is the node that should be used when following the witness links.

In order to find the real root of the equivalence class, the algorithm maintains for each PathFind representative the field "realroot". This points to the real root of the equivalence class, and it is the "witnesslink" from this real root that must be followed next. This field should be initialized for each node to point to the node itself.

In Figure 9, we show the effect of two simple MultiCertify operations. Note that the data structure contains three witness equivalence

```
MultiCertify(xx, yy):
¢  Assuming that xx and yy are equal, output a list of previously ¢
¢  asserted equalities to certify their equality.                ¢
begin
      x := Find(xx) ; y := Find(yy)
;     ToCertify := {(x, y)}
;     while ToCertify is not empty
      do    Remove an element (x, y) from ToCertify
      ;     x := PathFind(x) ; y := PathFind(y)
      ;     x := x.realroot ; y := y.realroot
      ;     x1 := x ; y1 := y
            ¢  Find the lowest common ancestor (lca) on the witness tree ¢
            ;    while ¬ x1.marked and ¬ y1.marked and x ≠ y
                 do
                      if   x1.witnesslink ≠ nil
                      then x1.marked := true
                      ;    x1 := PathFind(x1.witnesslink) ; x1 := x1.realroot
                      fi
                 ;    if   y1.witnesslink ≠ nil
                      then y1.marked := true
                      ;    y1 := PathFind(y1.witnesslink) ; y1 := y1.realroot
                      fi
                 od
            ;    if   x1.marked or x1 = y1
                 then lca := x1 ; overshoot := y1
                 else lca := y1 ; overshoot := x1
                 fi
            ¢  Clear any marks above the lowest common ancestor      ¢
            ;    t := lca
            ;    while t ≠ overshoot
                 do   t.marked := false
                 ;    t := PathFind(t.witnesslink) ; t := t.realroot
                 od
      ;     for t in {x, y}
            do   while t ≠ lca
                 do   Output t.witnesslabel ; t.marked := false
                 ;    t1 := PathFind(t.witnesslink) ; t1 := t1.realroot
                 ;    u := PathUnion(t, t1) ; u.realroot, t := t1, t1
                 od
            od
      od
end
```

Figure 8: Algorithm MultiCertify

classes: {e, d, f}, {a} and {g, b, c}. Note also that a representative node in the simple equivalence relation is not necessarily a representative in the witness equivalence relation. In Figure 10, we show the data structure of Figure 9 after one more MultiCertify operation. Observe that the equivalence classes have all collapsed into one. As all the witness edges in the data structure have now had their labels emitted, any further MultiCertify operation will have no effect.

3. An Overview of the Equality Algorithms

The presentation in this section has been adapted from that presented in chapter 6 of [Johnson 1981] with the author's permission. For a complete presentation of this algorithm and a detailed proof of its correctness, the reader is referred to [Johnson 1981].

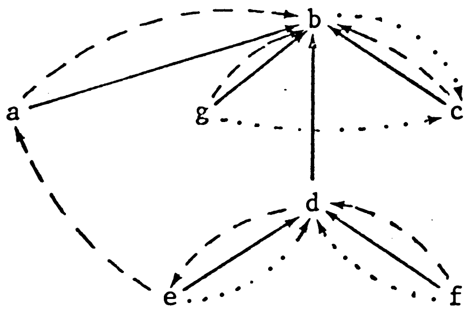

Figure 9: Data structure from Figure 6 after MultiCertify(g,c) and MultiCertify(e,f)
Dashed lines indicate witness links, solid lines indicate representative links. Witness representative links are shown as dotted lines.

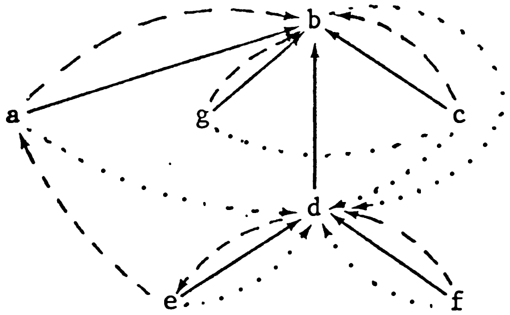

Figure 10: Data structure from Figure 9 after MultiCertify(f,c)

3.1. A Formal Definition of the Problem

Suppose we are given an alphabet A of function symbols. The syntactic object "expression" can be inductively defined as follows. If f is an element of A, and x_1, x_2, ... x_k are zero or more expressions, then f(x_1, x_2, ... x_k) is also an expression. The only expressions are objects that can be obtained by repeated applications of the preceding rules.

As in the simple variable case we let EQS represent the sequence of ordered pairs of expressions that have been declared to be "equal." Adding the pair {x, y} to EQS is an indication that in any context where x appears, y would do equally well. Two expressions are defined to be "equivalent" whenever one can be transformed to the other by a sequence of substitutions of expressions for their equals. (Observe that reflexivity will be represented by the null substitution sequence.) The equivalence of x and y in this sense is expressed as $x =_s y$. The input to Algorithm Eq is a series of instructions of the form "Assert(x,y)" and "Query(x,y)". The operation of algorithm Eq on these instructions is described in terms of the set EQS. (EQS is not used in the computations; it is merely used for specification). Initially, EQS is empty. When an Assert(x,y) instruction is encountered, the pair {x,y} is added to EQS. When a Query(x,y) instruction is encountered, a boolean value is output indicating whether $x =_s y$, where $=_s$ is the relation defined by the current value of EQS.

The problem of computing $=_s$ from EQS is equivalent to the uniform word problem for finitely presented algebras. Several algorithms for solving this problem have appeared in the literature. [Nelson and Oppen 1977] and [Downey, Sethi and Tarjan 1980] give algorithms that run in O(N log N) time. However, these algorithms are offline, which means that all the Assert instructions must come before all the Query instructions. [Samet 1980] gives an online solution to the problem, but does not give a complete runtime analysis. He simply states that the worst case runtime (his algorithm uses hashing) for a single Assert instruction is quadratic. Algorithm Eq processes a list of instructions in expected time of O(N log N) (it also uses hashing), where N is the total size of all the expressions in the input.

3.2. An Outline of Algorithm Eq

A linked data structure is used to represent expressions. Routine EqFind(e) is used to obtain the linked representation of the expression e. If e is of the form f(x_1, x_2, ... x_k), EqFind first calls itself recursively to obtain a node n_i representing each x_i. The node representing e contains the label f and pointers called operand links to the nodes n_1, n_2, ... n_j. A hashed search is used to determine if the data structure already contains such a node; if not, a new one is created.

The data structure is actually a directed acyclic graph, in which each node represents an expression. For example, if EqFind is called with arguments a(c(),d(g(),h())), b(d(g(),h()),f()) and e(h(),i()), the result will be the graph in Figure 11.

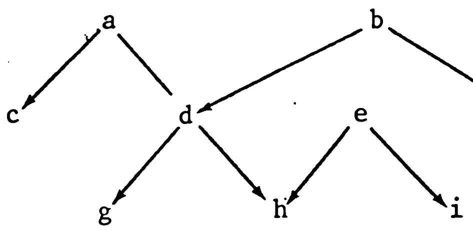

Figure 11: A graph created by EqFind

The asserted equality of two expressions e_1 and e_2 is recorded in the graph by connecting the node representing e_1 to the node representing e_2 using a pointer "reprlink". The nodes of the graph are partitioned into Union-Find equivalence classes, where all the nodes of a given equivalence class are connected by reprlinks to form a tree. One node in each equivalence class, the root of the tree, is singled out as the "representative" of the class. This node uniquely identifies the equivalence class.

Operand links are allowed to point only to representative nodes. Two equivalence classes are merged by connecting their representatives n_1 and n_2 with a reprlink. One of the nodes, say n_1, is chosen as the representative of the new equivalence class. All the operand links pointing to n_2 must be located and changed to point to n_1.

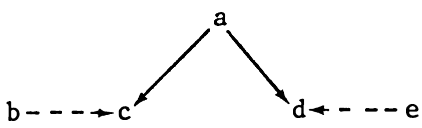

Figure 12: A graph with reprlinks

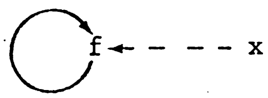

Figure 13: A graph with a cycle

A node represents a class of equivalent expressions. For example, the node with label a in the graph in Figure 12 (reprlinks are shown as dashed lines in figures) represents the following set of formulas:

{a(b(),d()), a(b(),e()), a(c(),d()), a(c(),e())}.

Once reprlinks are introduced, the graph is no longer constrained to be acyclic. Nodes in graphs with cycles can represent an infinite set of expressions. For example, the graph in Figure 13 is the result of asserting x and f(x) to be equal. The node with label x represents the infinite set {x(), f(x()), f(f(x())), f(f(f(x()))), ...}.

A Query(x,y) instruction is processed by testing whether the two expressions x and y are represented by the same node. For this strategy to work, it is crucial that enough reprlinks are added so that all nodes representing the same expression are in the same equivalence class.

Consider the graph in Figure 14a. As the result of the instruction Assert(d(), e()), the nodes representing d() and e() are "merged" by adding the reprlink and moving the operand links as shown in Figure 14b. But now the two nodes with b as a label, which are both representative nodes, have identical operand links. Only one of these nodes can be used as the representative of, for example, b(d(),e()). Since the two nodes currently have equal weight, that is they are the representatives of the same number of nodes (one each), the node on the right is arbitrarily chosen as the representative. The two nodes are then merged, as shown in Figure 14c. But now nodes that have the label a are equivalent and must be merged, as shown in Figure 14d.

The problem is to find all the nodes that need to be merged. Algorithm Eq maintains another set of links, called container links, that are used to locate the nodes that need to be updated when a reprlink is added.

3.3. The Details of Algorithm Eq

Algorithm Eq consists of the routines shown in Figures 15-18 and a global data structure, or "graph", as it will be called. The graph has three parts, the first of which is a collection of "nodes". The fields of the node are listed below.

<u>field</u>	<u>contents</u>
label	An input function name.
operands	A list of pointers to the nodes representing the operands for this function node.
containers	A list of pointers to operand fields of the nodes for which this is an operand. The operands and containers fields are heads of singly-linked lists of nodes. For the purposes of this discussion, we will assume that nodes are flexible, and the operand and container fields contain variable-sized lists.
reprlink	either a pointer to a node, or <u>nil</u> (marking the root) such that for all nodes x and y, x and y are in the same

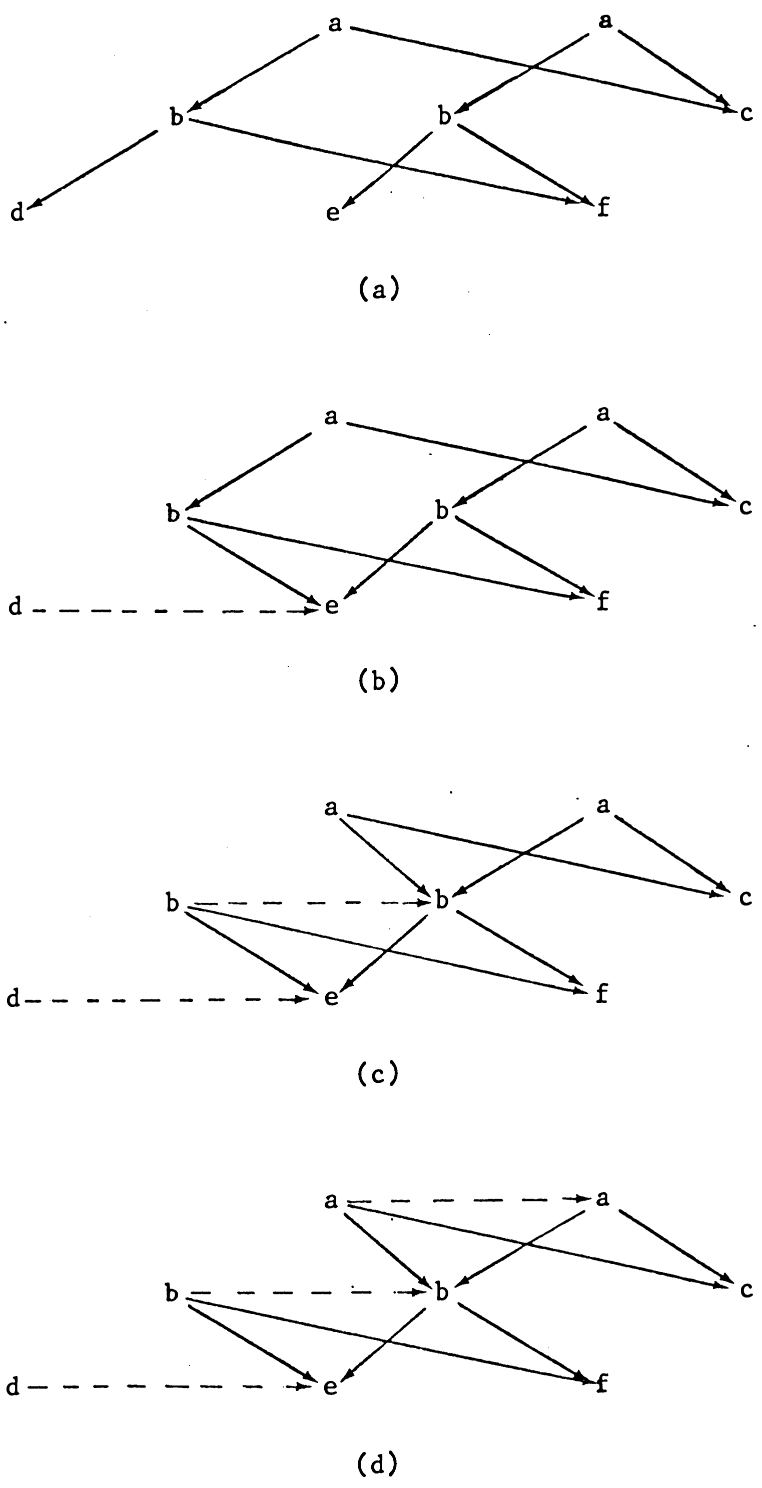

Figure 14: Merging equivalent nodes

```
Eq:
¢ Process a list of instructions ¢
begin
     while there are instructions left
     do    Read an instruction i
     ;     if    i is of the form Assert(x, y)
           then  EqAssert(x, y)
           elif  i is of the form Query(x, y)
           then  Output EqQuery(x, y)
           fi
     od
end
```

Figure 15: Algorithm Eq

```
EqQuery(xx, yy):
¢ Determine if expressions xx and yy are equal ¢
begin
       nx := EqFind(xx)
;      ny := EqFind(yy)
;      EqQuery := (nx = ny)
end
```

Figure 16: Algorithm EqQuery

	equivalence class if and only if their reprlinks lead to the same root node.
reprcount	the number of nodes in this node's equivalence class (if this node is a representative).
next	the next entry in this hash bucket.
previous	the previous entry in this hash bucket.
eligible	a boolean value, = "this node is in a hash bucket".

The term "node" will be used to refer to an area of storage, and not its contents. For example, if n_1 and n_2 are nodes, $n_1 \neq n_2$ means that n_1 and n_2 occupy different locations in memory, but does not exclude the possibility that corresponding fields of n_1 and n_2 have identical contents. Initially, there are no nodes. As the algorithm is executed, new nodes can be allocated, but nodes are never deallocated.

The second part of the graph is "bucket," an array of pointers to nodes. Each element of bucket is the head of a list of nodes that is doubly linked through the next and previous fields of each node. If a node is in a bucket, the node will be said to be "eligible"; otherwise it is said to be "ineligible". A bit in each node is used to keep track of whether the node is eligible; the bit is set when a node is put in a bucket, and cleared when it is removed.

```
EqFind(e):
¢ Add expression e to the graph ¢
begin
     e will be of the form f(x_1, x_2, ... x_k)
        for a function symbol f and some k >= 0
;    for each x_i do a[i] := EqFind(x_i) od
;    h := hash(f, (a[1], a[2], ... a[k]))
;    Search bucket[h] for a node with operator f
        and operands a[1], a[2], ... a[k]
;    if  the search succeeds in finding a node n
     then while n.reprlink ≠ nil do n := n.reprlink od
     else n := a pointer to a newly allocated node
     ;    Put n in bucket[h]
     ;    n.label := f
     ;    n.operands := (a[1], a[2], ... a[k])
     ;    Add a pointer to the i'th operand field of n
             to the container list of a[i], for each i in {1 ... k}
     ;    n.containers := the empty list
     ;    n.reprlink := nil
     ;    n.reprcount := 1
     fi
;    EqFind := n
end
```

Figure 17: Algorithm EqFind

Function "hash" takes as input a label and a list of the addresses of zero or more nodes and produces the index of a bucket. Initially, there are no nodes, so that all the buckets are empty.

The third part of the graph is "ToDo," which is a set of pairs of pointers to nodes. The notation "$\{n_1, n_2\}$ is in ToDo" will be defined to mean that either (n_1, n_2) or (n_2, n_1) is in ToDo. Initially, ToDo is empty.

4. A Decision Procedure for Equality with Certification

In this section we show how to combine the simple certification algorithm of section 2 with the equality decision procedure of section 3 to produce the complete algorithm.

The major problem in combining the two previous algorithms lies in dealing with operands. In adding operands, we have added a new equality axiom for congruence. In the congruence closure algorithm, we apply this axiom when two expressions have the same outermost operator, and their operands become equal. This occurs at line 27 of the EqAssert Algorithm (Figure 18).

```
EqAssert(xx, yy):
¢ assert the equality of expressions xx and yy ¢
begin
     nx := EqFind(xx) ; ny := EqFind(yy)
 ;   Add (nx, ny) to ToDo
     ¢ add {xx, yy} to EQS ¢
 ;   EqLoop:
     while ToDo is not empty
     do   Remove an element (x, y) from ToDo
     ;    while x.reprlink ≠ nil do x := x.reprlink od
     ;    while y.reprlink ≠ nil do y := y.reprlink od
     ;    if   x ≠ y
          then
               ¢ Let x be the name of the new representative ¢
                    if y.reprcount > x.reprcount
                    then x, y := y, x fi
          ;    y.reprlink := x
          ;    x.reprcount := x.reprcount + y.reprcount
          ;    ConLoop:
               for every w on y.containers
               do  w.operand := x
               ;    z := a pointer to the node containing w.operand
               ;    if z is eligible
                    then Remove z from the bucket it is in
                    ;    h := hash(z.label, z.operands)
                    ;    Search bucket[h] for a node with
                            the same label and operands as z
                    ;    if   the search finds a duplicate d
                         then Add (d, z) to ToDo
                         else Put z in bucket[h]
                         fi
                    fi
               od
          ;    Append y.containers to x.containers
          fi
     od
end
```

Figure 18: Algorithm EqAssert

How should we certify an equality that comes about by this congruence closure process? Since the certificates do not explicitly include references to reflexivity, we need only produce certificates for the equal pairs of operands. There is a cost involved in applying the congruence axioms, however, even if they do not show up explicitly in the certificate. We will consider this cost when calculating the running time of the algorithm.

We use normal witness edges to connect expressions that the congruence closure algorithm has internally declared to be equal. To identify these witness edges to the certification algorithm, we introduce a unique witness label, "equaloperands". When this label is encountered during a certify, the algorithm knows that it must certify the pairwise equality of the original operands of the two expressions.

Below, we list the fields for a node in the complete algorithm.

field	contents
label	An input function name.
operands	A list of pointers to the representative nodes of the operands for this function node.
originalops	A list of pointers to the nodes that were the actual original operands for this function.

```
ExtEq:
¢ Process a list of instructions ¢
begin
     while there are instructions left
     do    Read an instruction i
     ;     if    i is of the form Assert(x, y)
           then  ExtAssert(x, y)
           elif  i is of the form Query(x, y)
           then  Output ExtQuery(x, y)
           elif  i is of the form Certify(x, y)
           then  Output ExtCertify(x, y)
           fi
     od
end
```

Figure 19: Algorithm ExtEq

```
ExtQuery(xx, yy):
¢ Determine if expressions xx and yy are equal ¢
begin
      x := ExtFind(xx)
      while x.reprlink ≠ nil do x := x.reprlink od
;     y := ExtFind(yy)
      while y.reprlink ≠ nil do y := y.reprlink od
;     ExtQuery := (x = y)
end
```

Figure 20: Algorithm ExtQuery

containers A list of pointers to operand fields of the nodes for which this is an operand representative.

reprlink Either a pointer to a node, or nil (marking the root) such that for all nodes x and y, x and y are in the same equivalence class if and only if their reprlinks lead to the same root node.

reprcount The number of nodes in this node's equivalence class (if this node is a representative).

witnesslink Either a pointer to a node, or nil. If it is a pointer to a node, then the input contained a proclaim equating this node and the other node.

witnesslabel The index of the proclaimed equality that caused the witnesslink to be introduced or the distinguished label "equaloperands". If witnesslink is nil, this value is undefined.

witnessrepr Within the ExtCertify routine, this is either a pointer to a node or nil, such that for all nodes x and y, x and y are in the same witness equivalence class if and only if their witnessreprs lead to the same root node.

witnesscount If this node is a witness representative, this is the number of nodes in its witness equivalence class. (Used only within the ExtCertify routine.)

realroot Within the ExtCertify routine, if this node is the representative of a witness equivalence class, this is a pointer to the "real" root of that equivalence class. Initialized to point to the current node. (See text for a fuller explanation).

next Points to the next entry in this hash bucket.

previous Points to the previous entry in this hash bucket.

orignext Points to the next entry in the original hash bucket.

origprevious Points to the previous entry in the original hash bucket.

eligible a boolean value, = "this node is in a hash bucket".

marked false (used only within the ExtCertify procedure).

Except for the fields "originalops", "orignext" and "origprevious", these fields are simply the obvious combination of those given in sections 2 and 3. The new fields are needed to certify the equality of expressions linked by "equaloperands" witness links. In the congruence closure algorithm described in section 3, the operand list for an operator was continuously modified. This was done so that the operands of a node were always the representatives of the original operands. To process "equaloperands" witness links, the algorithm must be able to find the original operands of the expressions involved, in order to recursively certify their equality. The "originalops" field provides a list of these operands.

```
ExtFind(e):
¢ Return index of expression e in the graph (possibly inserting it) ¢
begin
     e will be of the form f(x1, x2, ... xk)
        for a function symbol f and some k >= 0
;    for each xi do b[i] := ExtFind(xi) od
;    oh := hash(f, (b[1], b[2], ... b[k]))
;    Search origbucket[oh] for a node with operator f
        and operands b[1], b[2], ... b[k]
;    if   the search does not succeed in finding a node n
     then n := a pointer to a newly allocated node
     ;    Put n in origbucket[oh]
     ;    n.label := f
     ;    n.witnessrepr := nil ; n.witnesscount := 1
     ;    n.originalops := (b[1], b[2], ... b[k])
     ;    for each i in {1 ... k}
          do   a[i] := b[i]
          ;    while a[i].reprlink ≠ nil do a[i] := a[i].reprlink od
          od
     ;    h := hash(f, (a[1], a[2], ... a[k]))
     ;    Search bucket[h] for a node with operator f
             and operands a[1], a[2], ... a[k]
     ;    if   the search succeeds in finding a node m
          then rm := m
          ;    while m.reprlink ≠ nil do m := m.reprlink od
          ;    n.reprlink := m ;  m.reprcount := m.reprcount + 1
          ;    n.witnesslink := rm ; n.witnesslabel := "equaloperands"
          else Put n in bucket[h]
          ;    n.operands := (a[1], a[2], ... a[k])
          ;    Add a pointer to the i'th operand field of n
                  to the container list of a[i], for each i in {1 ... k}
          ;    n.containers := the empty list
          ;    n.reprlink := nil ; n.reprcount := 1
          ;    n.witnesslink := nil ; n.witnesslabel := ""
          fi
     fi
;    ExtFind := n
end
```

Figure 21: Algorithm ExtFind

```
ExtAssert(xx, yy):
¢  Assert the equality of expressions xx and yy ¢
begin
          Equate_index := Equate_index + 1
          nx := ExtFind(xx) ; ny := ExtFind(yy)
    ;     Add (nx, ny, Equate_index) to ToDo
          ¢  Add {xx, yy} to EQS ¢
    ;     EqLoop:
          while ToDo is not empty
          do    Remove an element (rx, ry, witness) from ToDo
          ;     x := rx ; y := ry
          ;     while x.reprlink ≠ nil do x := x.reprlink od
          ;     while y.reprlink ≠ nil do y := y.reprlink od
          ;     if   x ≠ y
                then
                          ¢ Let x be the name of the new representative ¢
                                if y.reprcount > x.reprcount
                                then swap x and y fi
                     ;    y.reprlink := x
                     ;    x.reprcount := x.reprcount + y.reprcount
                          ¢ Reverse the witness path (ry,...,old representative) ¢
                          ;     w, previouslabel, v := ry,
                                          w.witnesslabel, w.witnesslink
                          ;     v := w.witnesslink
                          ;     while v ≠ nil do
                                ;     previouslabel, v.witnesslabel :=
                                                v.witnesslabel, previouslabel
                                ;     v.witnesslink, w, v := w, v, v.witnesslink
                                od
                          ;     ry.witnesslabel := witness ; ry.witnesslink := rx
                     ;    ConLoop:
                          for every w on y.containers
                          do  w.operand := x
                          ;     z := a pointer to the node containing w.operand
                          ;     if z is eligible
                                then Remove z from the bucket it is in
                                ;     h := hash(z.label, z.operands)
                                ;     Search bucket[h] for a node with
                                            the same label and operands as z
                                ;     if   the search finds a duplicate d
                                      then Add (d, z, "equaloperands") to ToDo
                                      else Put z in bucket[h]
                                      fi
                                fi
                          od
                     ;    Append y.containers to x.containers
                fi
          od
end
```

Figure 22: Algorithm ExtAssert

A related problem arises when an expression is being inserted into the data structure. Again, the algorithm must be able to find the original operands of the expression. The "orignext" and "origprevious" fields are simply links in the hash table "origbucket". The "origbucket" hash table is used in exactly the same way "bucket" is used in the simple congruence closure algorithm, but instead of accessing the expression based on the current representatives of its operands, it accesses the expression based on its original operands.

In Figures 19-23 we give the complete algorithm for the decision procedure for equality with logical dependency. In comparing these algorithms with their counterparts in section 3, note that the ExtFind function returns a pointer to the expression itself, and not to its representative as EqFind does. This is done so that the ExtCertify function can use ExtFind to find the expressions to be certified. Obviously, when doing a certify, the original expressions, and not the representatives, must be returned. Aside from this difference, the "Ext" algorithms are functionally equivalent to the "Eq" algorithms when deciding equalities.

The ExtCertify algorithm is shown in Figure 23. The need to certify arbitrary pairs of operands while still emitting a minimal set of certificates requires that we incorporate the MultiCertify algorithm of Figure 8 rather than the simpler Certify algorithm of Figure 4.

Again, as in section 2.2, the ExtCertify operation is destructive, in that it changes values in the data structure, and does not bother to reset them on exit. We assume that some sort of audit trail is being kept during the execution of ExtCertify, and on exit all values are restored to their values before ExtCertify was called.

As in section 2.2, the PathFind and PathUnion functions implement a weighted, path-compressing Union-Find. The "witnessrepr" and "witnesscount" fields are used by these functions. Note that this Union-Find operation is completely independent of other operations on the data structure. Only the PathFind and PathUnion functions will affect the "witnessrepr" and "witnesscount" fields, and, at the end of a call to ExtCertify, these fields are assumed to be reset to nil and zero respectively.

5. Correctness of the Algorithm

In this section we argue for the correctness of the logical dependency algorithm described in this report. We present an informal argument, and do not attempt to approach the level of detail of [Johnson 1981]. In particular, we argue for correctness only for the general algorithm. We will not be concerned with the details of the programs given in the sections above.

We need to make two basic arguments about the certificates produced by our algorithm. First, we must show that the queried equality can be deduced from the list of equalities in the certificate, using only the substitution and reflexive axioms for equality. Second, we must show that the certificate is minimal, i.e. that if we remove any equality from the certificate, then it is no longer possible to deduce the certified equality. In the arguments below, we will take as given that the

```
ExtCertify(xx, yy):
¢  Assuming that xx and yy are equal, output a list of previously ¢
¢  asserted equalities to certify it.                                ¢
begin
         x := ExtFind(xx) ; y := ExtFind(yy)
;        Add (x, y) to ToCertify
;        while ToCertify is not empty
         do    Remove an element (x, y) from ToCertify
         ;     x := PathFind(x) ; y := PathFind(y)
         ;     x := x.realroot ; y := y.realroot
         ;     x1 := x ; y1 := y
               ¢  Find the lowest common ancestor (lca) on the witness tree ¢
               ;   while ¬ x1.marked and ¬ y1.marked and x ≠ y
                   do
                        if   x1.witnesslink ≠ nil
                        then x1.marked := true
                        ;    x1 := PathFind(x1.witnesslink) ; x1 := x1.realroot
                        fi
                        if   y1.witnesslink ≠ nil
                        then y1.marked := true
                        ;    y1 := PathFind(y1.witnesslink) ; y1 := y1.realroot
                        fi
                   od
               ;   if   x1.marked or x1 = y1
                   then lca := x1 ; overshoot := y1
                   else lca := y1 ; overshoot := x1
                   fi
               ¢  Clear any marks above the lowest common ancestor     ¢
               ;   t := lca
               ;   while t ≠ overshoot
                   do   t.marked := false
                   ;    t := PathFind(t.witnesslink) ; t := t.realroot
                   od
         ;     for t in {x, y}
               do  while t ≠ lca
                   do   t.marked := false
                   ;    if t.witnesslabel = "equaloperands"
                        then Let the original operands of t.realroot be
                                 a[1], a[2], ... a[k] and let the original operands
                                 of t.witnesslink be b[1], b[2], ... b[k]
                        ;    for  each i in {1 ... k}
                             do   Add (a[i], b[i])) to ToCertify od
                        else Output t.witnesslabel
                        fi
                   ;    t1 := PathFind(t.witnesslink) ; t1 := t1.realroot
                   ;    u := PathUnion(t, t1) ; u.realroot, t := t1, t1
                   od
               od
         od
end
```

Figure 23: Algorithm ExtCertify

congruence closure algorithm correctly implements a decision procedure for equality.

5.1. Why are Certificate Equalities Sufficient for the Deduction?

We claim that the equality certificate for two expressions gives us a "program" of ExtAssert commands that will, if all the steps in the program are executed, put the two expressions into the same congruence closure equivalence class. Ignoring for a moment the problem of operands, if we begin at one of the queried expressions (at one end of the certificate) and follow the witness path (equalities in the certificate) doing ExtAsserts, each such assert puts the next expression on the path into the single growing equivalence class. The final step will put the second queried expression into the equivalence class. Since the congruence closure algorithm correctly implements a decision procedure for equality, the two expressions must therefore be equal simply on the basis of the equalities in the certificate.

The presence of operands changes the picture only slightly. When ExtAssert establishes an equality on the basis of equal operands, an "equaloperands" witness link is inserted. When ExtCertify encounters this witness link, the original operand pairs of the two expressions are placed on the ToCertify list (Figure 23, line 42). On some later iteration of the major loop in ExtCertify, these operand pairs will themselves be certified, and eventually all required certificates will have been emitted.

What about the Union-Find on witnesses? It can only cause trouble if a PathFind operation prevents the algorithm from emitting a witness label that would have otherwise been emitted. We claim that the labels on all witness edges internal to one of these equivalence classes have already been emitted. The only time a PathUnion is performed in ExtCertify is at line 46. Immediately prior to that line is an _if_ statement. Taking either branch of this _if_ statement causes the witness label to be effectively emitted from the edge on which the PathUnion is about to be performed. At this point it should be clear that PathFind operations can never cause ExtCertify to skip over unemitted witness labels.

5.2. Why Do the Certificate Equalities Form a Minimal Set?

The set of certificate equalities is minimal because removing any one of them will partition the equivalence class in such a way that the two expressions being certified will be in different equivalence classes. As in the section above, we will initially argue in the case of the simple Certify (without operands or witness Union-Find) and then examine the more complex case.

Suppose some equality in the certificate could be removed. This would correspond to removing a witness edge in the congruence closure graph. Since the witness edges form a tree, this would leave the two expressions at the ends of the certificate unconnected by witness edges. But the witness edges form a spanning tree on the congruence closure equivalence class, so that this would imply that the two expressions are in different equivalence classes. This contradicts the initial assumption that they were equal. We conclude that no equality in the certificate can be removed, and the certificate therefore forms a minimal set.

What happens when we include operands? It should be obvious that the case here is exactly that of the MultiCertify problem of section 2.2. That is, we can consider the problem of certifying operand equality as simply the problem of certifying a set of simple equalities.

The only problem we can encounter with the set of equalities is that of emitting duplicates. By exactly the argument we used above in the simple case, no equality that is deducible from the other equalities in a certificate (but is not equal to any of them) can ever appear in a certificate. Therefore, any superfluous equality in the certificate of the set of queried equalities must duplicate some other equality in the certificate.

The witness Union-Find guarantees that no duplicates will appear in the final output certificate. The argument for this is exactly that we gave for correctness at the end of section 5.2, with the added observation that the PathUnion operation is <u>always</u> performed immediately after the edge is emitted.

5.3. **Can We Do Better Than Minimal?**

The certificate produced by our algorithm is minimal, in the sense that removing any element makes it impossible to deduce the desired equality. However, it is not necessarily a minimum, or smallest possible, certificate. For example, if we ExtAssert a=b, a=c, and then b=c, the b=c equality will be deducible from the previous equalities, and hence will have no effect on the data structure. If we query b=c, our algorithm will produce the certificate {a=b, a=c}, even though the certificate {b=c} is obviously smaller. We might hope to find an algorithm that always produced a minimum certificate. Unfortunately, such an algorithm would allow us to solve an NP-complete problem.

In the problem of Steiner trees in graphs (problem [ND12] in [Garey and Johnson 1979], restricted to the case of equal edge weights) we are given a graph G = (V,E) and a subset R of V. We are asked to find a subtree of G including all the vertices of R such that the number of edges in the subtree is no more than some constant B. The mapping to our problem is straightforward. The vertices (V) of the graph are expressions, and the edges (E) are asserted equalities. For a set of connected vertices R = {a, b, c, d, ...}, the question we ask is: "Does f(b, f(c, f(d, f(...))) = f(a, f(a, f(a, f(...)))". The minimum set of equalities solving this problem forms a tree of minimum size, thus providing a solution to the Steiner tree problem.

Occasionally the hard part of a problem can be factored out as an isolated, unusual case (see [Chan 1977] for an example). We believe that in this problem no such factoring can be done, and the problem we are solving is inherently difficult.

6. **Running Time of the Algorithms**

On the question of running time, we wish to argue for two points. First, our changes to the congruence closure algorithm do not affect its basic running time. Second, the certification algorithm runs in time almost-linear in the number of equalities in the certificate plus the number of applications of congruence axioms required. We could explicitly list applications of the congruence axioms in the certificate, but they are so obvious that this seems undesirable.

In discussions of running time below we will take N to be the total number of characters in the input to ExtEq. The expected running time of congruence closure is taken to be O(N log N). (The algorithm uses hashing, so of course the worst case running time may be greater. For a complete discussion of the running time of the congruence closure algorithm, the reader is referred to [Johnson 1981].)

6.1. Changes to the Running Time of Congruence Closure

Only two of our changes to the congruence closure algorithm involve loops or recursion. These are the only changes that can affect the running time by more than a constant factor.

The first such change is the addition of a constant number of representative lookups to ExtFind (Figure 21, lines 16-19) (note that the number of operands of a function is assumed to be bounded by some constant). Each of these lookups can cost at most O(log N), and ExtFind can be called at most O(N) times. Therefore the total added cost here will be O(N log N), which will affect the overall running time of congruence closure by only a constant factor.

The second change involving a loop is somewhat more subtle. In the ExtAssert algorithm (Figure 22), code has been added to the basic congruence closure algorithm to reverse the witness path from a node to its representative in the graph. This code is at lines 21-29. Since the length of the witness path from a node to its representative can reach the size of the equivalence class, this step in the algorithm can take time up to O(N). However, each time this reversal is done, it is done in the smaller of two equivalence classes that are being merged. The operation is only done when two equivalence classes are merged, and the cost is O("size of the smaller class"). Each time this operation is done, the size of the smaller class is at least doubled. Therefore, this operation can be done at most O(log N) times. Thus, the total cost of all such operations is O(N log N). Once again, we have affected the overall running time of the congruence closure algorithm by only a constant factor.

6.2. Running Time of Certification

We have been claiming that we can do the certify operation in time almost-linear in the number of equalities in the certificate. This would be exactly correct if applications of the congruence axioms were listed in the certificate. To be precise, we will require time proportional to the sum of the number of applications of congruence axioms and the sizes of the equalities in the certificate times Tarjan's α function [Tarjan 1975] (a very slow-growing function related to the inverse of Ackermann's function) on this value. We will refer to the sum of the number of applications of the congruence axioms and the sizes of the certificate equalities as K.

We must consider each loop in the ExtCertify routine (Figure 23). The the number of iterations of the outermost loop (at line 7) is based on the total number of pairs put into ToCertify. The input expression pair to be certified is put into ToCertify at line 6. The only other time an expression pair is put into ToCertify is at line 42. With each pair of operands put into ToCertify at line 42 we can associate an application of the congruence axioms or a pair of operands in one of the certifying equalities. This outer loop will therefore be bounded by O(K). Moreover, the work done for each ToCertify pair is, except for PathFind operations, disjoint from the work done for any other pair.

The PathFind operations at lines 9, 17, 21, 32, and 45 will, over the O(K) iterations of the loops containing them take at most time $O(K\ \alpha(K))$. The loops at lines 13-23, 28-33, and 34-48 are each bounded by twice the length of the witness path between the two expressions, and thus by O(K). Moreover, since the different ToCertify pairs require disjoint effort, the total work done in these loops in all iterations through the outermost loop will be O(K). Thus, we see that the total cost of the algorithm will be bounded by $O(K\ \alpha(K))$, as desired.

7. Conclusion

We believe that the algorithm for determining logical dependency presented here is both practical and efficient. It is being incorporated into the AVID interactive proof development system, where it should be a useful proof development aid. The algorithm can potentially be included in any of several verification systems that make use of the congruence closure algorithm for deciding equality (e.g. the Stanford Pascal Verifier [Luckham et al 1979]).

References

[Aho, Hopcroft and Ullman 1974]
Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman. The Design and Analysis of Computer Algorithms, pages 124-139, Addison-Wesley, 1974.

[Chan 1977]
Tat-Hung Chan. An Algorithm for Checking PL/CV Arithmetic Inferences. Computer Science Department Technical Report 77-326, Cornell University, October 1977.

[Constable and Johnson 1979]
Robert L. Constable and Scott D. Johnson. A PL/CV Precis. Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages, pp. 7-20, January 1979.

[Constable and O'Donnell 1978]
Robert L. Constable and Michael J. O'Donnell. A Programming Logic. Winthrop, 1978.

[Downey, Sethi and Tarjan 1980]
Peter J. Downey, Ravi Sethi, and Robert E. Tarjan. Variations on the Common Subexpression Problem. Journal of the Association for Computing Machinery XXVII(4):758-771, October 1980.

[Garey and Johnson 1979]
Michael R. Garey and David S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. pages 208-209, W.H. Freeman, 1979.

[Johnson 1981]
Scott D. Johnson. A Computer System for Checking Proofs. Ph.D. Thesis, Department of Computer Science, Cornell University, January 1981.

[Kozen 1977]
Dexter Kozen. Complexity of Finitely Presented Algebras. Ph.D. Thesis, Department of Computer Science, Cornell University, May 1977.

[Krafft 1978]
Dean B. Krafft. The Assertion Table System for the PL/CV2 Program Verifier. Computer Science Technical Report 78-337, Cornell University, April 1978.

[Krafft 1981]
Dean B. Krafft. AVID: A System for the Interactive Development of Verifiably Correct Programs. Ph.D. Thesis, Department of Computer Science, Cornell University, in preparation, May 1981.

[Luckham et al 1979]
D. C. Luckham and the Stanford Verification Group. Stanford Pascal Verifier User Manual. Computer Science Department Report No. STAN-CS-79-731, Stanford University, March 1979.

[Nelson and Oppen 1977]
Greg Nelson and Derek C. Oppen. Fast Decision Algorithms Based on Union and Find. Proceedings of the Eighteenth Annual Symposium on

the Foundations of Computer Science, pages 114-119, IEEE Computer Society, 1977.

[Samet 1980]
Hanan Samet. Efficient On-line Proofs of Equalities and Inequalities of Formulas. IEEE Transactions on Computers C-29(1):28-30, January 1980.

[Tarjan 1975]
Robert E. Tarjan. Efficiency of a Good But Not Linear Set Union Algorithm. Journal of the Association for Computing Machinery XXII(2):215-225, April 1975.