TR-553 August 1977

EQUIVALENCE AND INEQUIVALENCE OF INSTANCES OF FORMULAS

by

Hanan Samet
Computer Science Department
University of Maryland
College Park, Maryland 20742

Abstract

An algorithm is presented for determining whether or not two instances of formulas are equal based on previous equality and inequality declarations. The problem is attacked using a formal grammar approach. The equality determination algorithm is shown to be almost linear, on the average, along with a completeness proof. The maximum space requirement for the equality data base is also discussed.

Keywords and phrases: equality, program verification, code optimization, theorem proving, hashing

CR categories: 3.70,4.29,5.21,5.24

1. INTRODUCTION

We are interested in a system to handle equality operations based on known equivalences and inequivalences of instances of formulas. Such applications arise in the domains of program verification ([London72],[Samet78]), code optimization [CockeSchwartz70], and theorem proving [ChangLee73]. Some of the requests to which we would like to be able to respond are:

(1) Are two items known to be equal?

(2) Are two items known to be unequal?

(3) Is it impossible to determine if two items are equal or not?

(4) Update the data base to include an additional equality.

(5) Update the data base to include an additional inequality.

(6) Does the inequality of two items lead to a contradiction (i.e., an implied equality)?

(7) Does the equality of two items lead to a contradiction (i.e., an implied inequality)?

A few examples of the type of requests made to the system are given below:

$$\begin{aligned}
&\text{Ex. 1:}\quad \text{Given:} & a &= b\\
&& c &= d\\
&& b &= c\\
&\phantom{\text{Ex. 1:}}\quad \text{Derive:} & g(a) &= g(d)\\[4pt]
&\text{Ex. 2:}\quad \text{Given:} & g(b) &= f(a)\\
&& g(c) &= f(b)\\
&& a &= b\\
&& c &= d\\
&\phantom{\text{Ex. 2:}}\quad \text{Derive:} & g(a) &= g(d)\\[4pt]
&\text{Ex. 3:}\quad \text{Given:} & c &= d\\
&& f(a) &= a\\
&& a &= c\\
&\phantom{\text{Ex. 3:}}\quad \text{Derive:} & f(f(a)) &= d\\[4pt]
&\text{Ex. 4:}\quad \text{Given:} & f(b) &= a\\
&& f(a) &= a\\
&& f(f(a)) &= c\\
&\phantom{\text{Ex. 4:}}\quad \text{Derive:} & f(f(b)) &= c\\[4pt]
&\text{Ex. 5:}\quad \text{Given:} & f(a,b) &\ \text{NEQ}\ f(c,d)\\
&& a &= c\\
&\phantom{\text{Ex. 5:}}\quad \text{Derive:} & b &\ \text{NEQ}\ d
\end{aligned}$$

Note that these examples, and all future examples, are written in a functional notation using parentheses and commas although our algorithm will be for formulas written in a prefix functional notation. In the remainder of the paper we shall present efficient algorithms to answer these questions using a formal grammar approach. For another treatment to similar problems using dags see [DowneySethi77].

2. OTHER APPROACHES

One way of keeping track of the equivalences is by means of equivalence classes [GallerFisher64]. When a new equality is seen, the current list of equivalences is updated to reflect all possible members based on the new equality. One pitfall of such an approach is that all possible equalities can not be generated since such a procedure will not terminate (e.g. f(a)=a will cause substitution to go on forever). A variation of this approach is to have pointers to all equivalence classes. When an equality is determined, all subexpressions of the equality pair that appear in previous equivalence classes have their respective pointers substituted. Next, the class name of the new equivalence class is substituted in all equivalence classes where members of the new equivalence appear as a subexpression. Another pitfall, which must be considered in all approaches, is the case when two items are known to be inequivalent, yet subsequent equality operations could cause a contradiction.

The last pitfall can be illustrated by examining a variant of example 5. In this case we could not assume, in addition to the two given equalities, that b=d since this would lead to a contradiction. Namely, a=c and b=d imply that f(a,b)=f(c,d) which is known to be false. Thus the fact that f(a,b) is known to be inequivalent to f(c,d) implies that (a NEQ c) or (b NEQ d) or both, but not equality (i.e., a=c and b=d).

Closer examination of example 4 will reveal that in effect f(f(b)) is being reduced to a during the process of trying to prove f(f(b))=c. This suggests a process of equivalence by reduction which is analogous to parsing. Thus, perhaps our problem can be specified in terms of grammars.

Given a grammar G for our language of formulas we wish to determine if two sentences of the language are equal based on a known set of equalities. The set of equalities is a set of pairs of strings that are sentences of the language generated by G (henceforth referred to as L(G)). The set of equalities can be considered to be a set of symmetric productions where each member of a production corresponds to a formula in prefix functional notation. Each argument position in a formula contains either another formula or a nonterminal symbol which corresponds to an atom. For example, consider fig. 1 where the representation of the equality f(a,b)=g(c) is demonstrated. Note the use of lower case letters for terminal symbols and upper case letters for nonterminal symbols.

```
(f A B)  ==>  (g C)
 (g C)   ==>  (f A B)
   A     ==>    a
   B     ==>    b
   C     ==>    c
```

Fig. 1 - Equality Grammar for f(a,b)=g(c)

Thus the problem can be formulated in terms of formal languages. Namely, there is a set of productions, G, containing two productions for each equality and one production for each terminal symbol. The problem of equality determination can now be reformulated as follows:

> Given a pair of sentences of L(G), determine if after reducing each terminal symbol of the first sentence, say S1, to its corresponding nonterminal symbol, the set of strings generated from this string of nonterminals contains the second sentence, say S2.

However, G is a type 0 grammar whose decision problem is undecidable. Therefore, the procedure of generating all sentences of a given length, inefficient as it might be, is impossible. The problem, formulated in such terms, is undecidable for the general case. One of the basic troubles of this approach is that it does not make use of either transitivity information or the syntax of L(G). In other words, transitivity must be rederived each time it is desired to check if two items are equal. Nevertheless, an advantage of such a method, if it were feasible, is that when new equality relations are determined, updating the data base consists of merely adding the symmetric productions.

3. SOLUTION

The procedure that we will use is based on some special properties of our grammar, G, given in fig. 2. Note the use of <atom> to indicate a literal name. Also there is one production for each function consisting of the function name, and as many S symbols as there are arguments expected by the function.

```
S ==> <atom>
S ==> (<fname1> S S . . . S)
S ==> (<fname2> S S . . . S)
                .
                .
                .
S ==> (<fnameN> S S . . . S)
```

Fig. 2 - Sample Equality Grammar

The basic problem to which we will address ourselves is that of creating and updating a data base for equalities so that a simple decision algorithm can be used to determine if two sentences of L(G) are indeed equal. We will use the notion of equivalence classes to keep track of all sentences known to be equal. An equivalence class is constructed for each valid sentence of L(G) which has been encountered while processing equalities of L(G). Moreover, the components of each valid sentence of L(G) are represented in terms of their equivalence classes. This is a crucial property of the system for it enables the representation to be recursive (i.e., the components of an equivalence class may refer to the class). In fact it is this property which distinguishes the system from one of the typical approaches mentioned earlier and enables us to represent such equalities as f(a)=a.

For example, if f(a)=f(b), then when this equality is processed an equivalence class is created for a (say A0), for f(a) (say A1 whose contents is f(A0)), for b (say A2), and for f(b) (say A3 whose contents is f(A2)). The equality of f(a) and f(b) is noted by merging the two equivalence classes A1 and A3, and all subsequent references to f(a) or f(b) are by use of the lowest numbered equivalence class which was merged - i.e . A1 in our case. As another example, suppose a=b, then all subsequent references to a or b are via their class name (i.e., A0). Thus if f(a) or f(b) were to occur in other sentences, then they would be represented by a unique equivalence class name whose contents is f(A0). A final example is the representation of f(a)=a . In this case a is identified by the equivalence class A0, while f(a) is identified by the equivalence class A1 whose contents is f(A0). The instance of equality is represented by the fact that all future references to a and f(a) are by their equivalence class name - i.e., A0.

Thus we see that our data base must include the various equivalence classes and their contents in terms of other equivalence classes. At this point it becomes clear that we have constructed an equality grammar, G, whose nonterminal symbols are the names of the equivalence classes and the productions are simply the equivalence class names deriving each of their respective members.

The process of adding an equality to our data base consists of:

(1) For each half of the equality determine the equivalence class in which it is contained (and the creation of one if it is not contained in any equivalence class).

(2) Merge the two equivalence classes.

(3) Update all references to the merged equivalence classes to point to the new equivalence class.

(4) Merge all equivalence classes whose equivalence is a direct consequence of 2.

As a clarification of (4) consider the case when a=b, and f(a) and f(b) appear in separate equivalence classes. Then (2) implies that f(a) and f(b) are to be uniquely represented as f(<ecln>) (<ecln> is the name of the equivalence class containing a=b), and thus the two classes containing f(a) and f(b) are merged. In other words, all equivalence classes are checked against each other for elements in common; and if yes, then a merge occurs and only one of the duplicate entries is kept in the newly formed equivalence class.

The process of determining the equivalence class containing a sentence of L(G) is the same as parsing a sentence of of L(G). The only difference is that instead of making a reduction to the nonterminal symbol S (and also the start symbol), we make a reduction to the appropriate equivalence class. If a reduction can not be made, then the sentence is not a member of any of the known equivalence classes, and a new equivalence class (containing only the sentence in question) is created and parsing continues. Reductions, if they exist, are always unique since any sentence is contained in only one equivalence class. In fact, the ability to add equivalence classes while parsing is what enables us to prove that f(a)=f(b) given that a=b.

Our algorithms take advantage of some special properties of G. The main problem in parsing is that the sentence being parsed could possibly not be a member of any equivalence class. This is equivalent to stating that there is a reduction to be made, yet there is no nonterminal symbol to which the handle is to be reduced. In our case the problem is somewhat alleviated by the fact that G is always simple precedence [Martin68] (see the proof in the appendix) and thus we always know when a reduction is desired. We take advantage of this situation by examining the equivalence classes and determining if a reduction exists. If yes, then the reduction is made and processing continues in a normal manner. If not, then it is known that the current handle (which is a sentence of L(G)) is not a member of any equivalence class, and thus a new class is created with the handle as its sole member. This is a natural extension to the process of creating a class for each atom not already in a class since atoms are also valid sentences of L(G). By atom we mean literal names and not function names. Note that since equivalence classes always contain valid sentences of L(G), the equivalence classes have the same precedence relations as S (the start symbol). Also, two equivalence class names can only have the precedence relation = between them, and in fact they always have this relation. The remaining precedence relations are given in the precedence matrix in fig. 3 (note the use of <ecln> to denote an equivalence class name).

	<atom>	<fname>	<ecln>	(	)
<atom>	>		>	>	>
<fname>	<		=	<	
<ecln>	<		=	<	=
(		=			
)	>		>	>	>

Fig. 3 - Precedence Matrix for Fig. 2

4. EQUALITY DETERMINATION ALGORITHM

The equality data base consists of a set of entries each of which is a sentence of L(G). Each entry is uniquely numbered, referred to as index below, and has a left part, which is the sentence value, and a right part which is a pointer to the sentence representing the equivalence class to which it belongs. A sentence value is either an atom or a list consisting of a function name followed by pointers to the equivalence classes containing the arguments of the function being represented by the sentence. Thus it is seen that each entry in the data base is a production:

index ==> left

The algorithm for adding an equality pair to the data base is given in fig. 4 using a combination of LISP [McCarthy60] and ALGOL [Naur60]. Note that all references to a member of an equivalence class are in terms of its head (i.e., the lowest numbered member of the equivalence class). The nature of adding entries to the data base, and the fact that when a merge occurs the head of the new equivalence class becomes the lowest numbered component of the merge insure that all sentence values are in terms of lower numbered equivalence classes. Moreover, by step 5 of the algorithm no sentence is included in more than one equivalence class, and each sentence appears only once in an equivalence class.

To add an equality pair:

```
1.  classl:=result of parsing left half;
2.  classr:=result of parsing right half;
3.  mods:=nil;
4.  a.  m:=min(classl,classr);
    b.  n:=max(classl,classr);
    c.  for j:=m+1 step 1 until maxclass do begin
              comment:  find candidates for implied equivalence;
          if null(right(eqtable[j])) then nil
            comment:  check if a deleted entry;
          else if atom(left(eqtable[j])) then nil
          else if member(m,cdr(left(eqtable[j]))) then mods:=merge(j,mods)
          else nil
          end;
    d.  for j:=n step 1 until maxclass do begin
              comment:  replace all instances of the higher numbered
                        equivalence by the lower numbered one;
          if null(right(eqtable[j])) then nil
            comment:  check if a deleted duplicate entry;
          else begin
            if right(eqtable[j]) = n then right(eqtable[j]):=m;
              comment:  change index to point to new head of equivalence
                        class;
            if atom(left(eqtable[j])) then nil
            else if member(n,cdr(left(eqtable[j]))) then begin
                comment:  replace higher numbered equivalence class by lower
                          numbered one;
              subst(m,n,cdr(left(eqtable[j])));
              mods:=merge(j,mods);
                comment:  add to list of candidates for implied equivalence;
              end
            else nil;
            end;
          end;
```

```
5.  while not null(mods) do begin
        comment:  step through the list of candidate nodes and look for
                  implied equivalences;
      for k:=2 step 1 until length(mods) do begin
        if left(eqtable[mods[1]]) = left(eqtable[mods[k]]) then begin
            comment:  an entry appears more than once in the equality data
                      base;
          temp:=right(eqtable[mods[k]]);
          mods:=delete(k,mods);
            comment:  delete the higher numbered duplicate entry;
          right(eqtable[mods[k]]):=nil;
            comment:  check if two entries are already in the same
                      equivalence class;
          if right(eqtable[mods[1]]) NEQ temp then begin
              comment:  two equivalence classes must be merged;
            classl:=temp;
            classr:=right(eqtable[mods[1]]);
            go to 4;
            end;
          end;
        end;
      mods:=cdr(mods);
      end;
```

Fig. 4 - Algorithm to Add an Equality Pair

In the algorithm eqtable is an array, accessed by left and right, which contains one entry for each formula. mods is a list sorted in ascending order containing pointers to entries in eqtable which refer to any members of merged equivalence classes. maxclass is the highest numbered equivalence class in eqtable. Note that the algorithm's implementation is far more efficient than its representation here since we have put the emphasis on clarity. For a more efficient representation see fig. 5 and also the discussion in section 7 on time and space requirements.

The algorithm terminates since parsing (steps 1 and 2) is a process that is limited by the length of the input string and by the number of productions. Step 4 is a merge of two equivalence classes and the time it takes is bounded by the number of productions. Step 5 is used to determine if a merge of two equivalence classes is to occur when a previous merge has caused two equivalence classes to have an element in common. If this is the case, then the two equivalence classes are merged and the resulting class has only one occurrence of the previously duplicate entry. In order to perform the merge the algorithm is reapplied. However, when the algorithm is reapplied we have one less equivalence class and thus by the well ordering principle termination is guaranteed. If no two distinct equivalence classes have elements in common, then mods is exhausted and we are through. Note that if an equivalence class is found to contain a duplicate occurrence of an element after a merge, then the duplicate occurrence is deleted from the class (in fig. 4 this is achieved by setting to NIL the right field of the eqtable entry to be deleted). This insures that our grammar will always have the property that no two productions have the same right hand side (i.e., an unambiguous grammar).

The motivation behind step 4 is the propagation of transitivity between equivalence classes while step 5 propagates transitivity via function application. In other words, functions of equal arguments are equal and thus their equivalence classes are merged. Steps 4c and 4d insert in mods all entries that are affected by the merge of equivalence classes - i.e., only these sentences refer directly to the two items whose equivalence classes have been merged. Note that steps 4c and 4d take advantage of the property that the head of an equivalence class is the lowest numbered entry in the class. Similarly, step 5 is only applied to entries in mods because only these entries can possibly generate new equivalences. This takes advantage of the fact that prior to the application of the algorithm the equivalence classes are disjoint, and thus implied equality between equivalence classes can only occur via elements pointing to the merged equivalence classes (this is an inductive argument).

If mods would not be sorted, then step 5 would not be as simple since there would be a question as to which of mods(l) or mods(k) should be deleted. The reason mods(k) is always deleted is that the equivalence class to which it belongs can not be both greater than mods(l) and also be the minimum of the classes containing mods(l) and mods(k). This is because the number associated with the head of an equivalence class is always less than the number associated with any of its component classes. Thus even when mods(k) is deleted, the property of the data base having all entries point to lower numbered entries is preserved.

Equality can be determined quite easily. We simply parse the two items in question in the following manner:

(1) Parse one item with the existing set of productions. This set is modified whenever a reduction is encountered which has no corresponding nonterminal symbol (i.e., the sentence is not a member of any equivalence class).

(2) Parse the second item with the modified grammar from part 1. If the two items are equal, then no modifications to the grammar will be necessary at this stage. In fact, if any modifications were made, then the items are not equal (i.e., it is impossible to determine if the two items are equal based on equality information at hand).

(3) If steps (1) and (2) yield identical equivalence class names, then the two items are equal. Otherwise, they are not known to be equal.

At this point we must prove that statement 3 is true. This is equivalent to the following theorem:

Theorem: The algorithm for determining equality is complete.

Proof: The theorem is a direct consequence of the following two lemmas:

Lemma 1: If the algorithm indicates that two items are equal, then they are equal.

Proof: This statement is true since by construction the equality updating algorithm insures that all elements in an equivalence class are equal.

Lemma 2: If two items are equal, then the algorithm will so indicate.

Proof: The proof of this statement reduces to showing that if two items are equal, then they will appear in the same equivalence class. This is proved by considering the two ways in which two items can be equal.

(a) The items were explicitly equal. In this case they will appear in the same equivalence class by virtue of step 4 of the updating algorithm and hence are always referred to by the new equivalence class name.

(b) The items became equal via transitivity and or functional application. In this case the items, say A and B, are equal to some third item C and now:

Either C must be in neither equivalence class which is impossible by step 5 of the updating algorithm which examines all transitivities and function application of equals.

Or C must be an element of both equivalence classes. This is impossible since this means that there exist two leftmost derivations of C thereby contradicting the unambiguousness of G which is true by virtue of G a being a simple precedence grammar. Therefore if two items are equal, then they will be in the same equivalence class.

Q.E.D.

Thus the above algorithm for determining equality is complete.

Q.E.D.

A more efficient implementation of the equality updating algorithm is given in fig. 5. It is different from the algorithm given in fig. 4 in that the number of passes over the table is reduced. In addition, it is made more suitable to a LISP-like implementation where the number of copy operations is to be minimized. Note that mods is now a sorted list in descending order, rather than ascending order, containing pointers to entries in eqtable which refer to any members of merged equivalence classes. Also we have added a parameter, dels, which is a sorted list in descending order containing the names of eqtable entries which are to be deleted. The purpose of dels is shown by step 7 which removes from eqtable all entries that step 8 has found necessary to delete. Upon exiting step 7, dels is guaranteed to be empty because its minimum entry is greater than or equal to n (i.e., max(classl,classr)) since n is the head of the class containing the minimum entry of dels (recall that step 8 proceeds from the last entry in eqtable towards the first entry). Step 9 is used to purge the class names in dels from eqtable when mods is exhausted (i.e., we are through and thus we will not return to step 7).

At this point we see that the updating algorithm can be described as incorporating one of the approaches mentioned earlier as a solution to the equality problem. Parsing is analogous to the process of substituting equivalence class names for all subexpressions known to be members of equivalence classes while steps 7 and 8 of fig. 5 are analogous to substituting the newly derived equality everywhere it appears as a subexpression.

```
To add an equality pair:

1.  class1:=result of parsing left half;
2.  classr:=result of parsing right half;
3.  mods:=nil;
4.  dels:=nil;
5.  m:=min(class1,classr);
6.  n:=max(class1,classr);
7.  for j:=maxclass step -1 until m+1 do begin
        comment:  perform all deferred deletions and substitutions, and
                  determine candidates for implied equivalence;
      if null(right(eqtable[j])) then nil
      else if j = dels[1] then begin
          comment:  delete entry j;
        right(eqtable[j]):=nil;
        dels:=cdr(dels);
        end
      else begin
        if right(eqtable[j]) = n then right(eqtable[j]):=m;
          comment:  replace occurrence of higher numbered equivalence class
                    by lower numbered one;
        if atom(left(eqtable[j])) then nil
        else if member(n,cdr(left(eqtable[j]))) then begin
            comment:  a candidate for implied equivalence;
          subst(m,n,cdr(left(eqtable[j])));
          mods:=merge(j,mods);
          end
        else if member(m,cdr(left(eqtable[j]))) then mods:=merge(j,mods);
          comment:  determine if a candidate for implied equivalence;
        end;
      end;
8.  while not null(mods) do begin
        comment:  step through the list of candidate nodes and look for
                  implied equivalences;
      len:=length(mods);
      for k:=2 step 1 until len do begin
        if left(eqtable[mods[1]]) = left(eqtable[mods[k]]) then begin
            comment:  an entry appears more than once in the equality data
                      base;
          if right(eqtable[mods[1]]) NEQ right(eqtable[mods[k]]) then begin
              comment:  the two entries are not in the same equivalence
                        class and the two equivalence classes must be
                        merged;
            class1:=right(eqtable[mods[k]]);
            classr:=right(eqtable[mods[1]]);
            dels:=reverse(mods[1] cons dels);
            mods:=cdr(mods);
            go to 5;
            end
          else begin
              comment:  the two entries are in the same equivalence class;
            k:=len;
              comment:  add mods[1] to the deferred deletion list and cease
                        searching for entries matching mods[1] - i.e., exit
                        the for loop;
            dels:=mods[1] cons dels;
            end
          else nil;
          end;
        end;
      mods:=cdr(mods);
      end;
9.  purge(dels,eqtable);
      comment:  remove all entries in dels from eqtable when no more implied
                equivalences;
```

Fig. 5 - A More Efficient Algorithm to Add an Equality Pair

5. EXAMPLES

In the following examples, each numbered line represents the result of either parsing a sentence or updating the data base to include a new equality. In the former case only the modifications to the data base, eqtable, are shown while in the latter case the entire updated data base is shown. Also in the former case the name of the equivalence class containing the sentence being parsed is returned.

As an example of the updating algorithm, consider example 4 of section 1.

sentence		eqtable			result
1. f(b)	yields	A0: A1:	b f(A0)	A0 A1	returns A1
2. a	yields	A2:	a	A2	returns A2
3. f(b)= a	yields	A0: A1: A2:	b f(A0) a	A0 A1 A1	
4. f(a)	yields	A3:	f(A1)	A3	returns A3
5. a	yields		no change		returns A1

6. f(a) = a	yields	A0:
		A1:
		A2:
		A3:
			b
			f(A0)
			a
			f(A1)
				A0
				A1
				A1
				A1	

7. f(f(a))	yields		no change		returns A1
8. c	yields	A4:	c	A4	returns A4
9. c = f(f(a))	yields	A0: A1: A2: A3: A4:	b f(A0) a f(A1) c	A0 A1 A1 A1 A1	

At this point we wish to determine if f(f(b)) = c

f(f(b)) gets parsed successively as:

1. f(f(A0))
2. f(A1)
3. A1

and c gets parsed as A1. Therefore, f(f(b)) = c .

As a more complicated example, consider example 2.

sentence		eqtable			result
1. g(b)	yields	A0: A1:	b g(A0)	A0 A1	returns A1

2.	f(a)	yields	A2:	a	A2	returns A3
			A3:	f(A2)	A3	
3.	g(b) = f(a)	yields	A0:	b	A0	
			A1:	g(A0)	A1	
			A2:	a	A2	
			A3:	f(A2)	A1	
4.	g(c)	yields	A4:	c	A4	returns A5
			A5:	g(A4)	A5	
5.	f(b)	yields	A6:	f(A0)	A6	returns A6
6.	g(c) = f(b)	yields	A0:	b	A0	
			A1:	g(A0)	A1	
			A2:	a	A2	
			A3:	f(A2)	A1	
			A4:	c	A4	
			A5:	g(A4)	A5	
			A6:	f(A0)	A5	
7.	a	yields		no change		returns A2
8.	b	yields		no change		returns A0
9.	a = b	yields	A0:	b	A0	
			A1:	g(A0)	A1	
			A2:	a	A0	
			A3:	f(A0)	A1	
			A4:	c	A4	
			A5:	g(A4)	A5	
			A6:	f(A0)	A5	
		followed by	A0:	b	A0	
			A1:	g(A0)	A1	
			A2:	a	A0	
			A3:	f(A0)	A1	
			A4:	c	A4	
			A5:	g(A4)	A1	
			A6:	f(A0)	NIL	
10.	c	yields		no change		returns A4
11.	d	yields	A7:	d	A7	returns A7
12.	c = d	yields	A0:	b	A0	
			A1:	g(A0)	A1	
			A2:	a	A0	
			A3:	f(A0)	A1	
			A4:	c	A4	
			A5:	g(A4)	A1	
			A6:	f(A0)	NIL	
			A7:	d	A4	

At this point we wish to determine if g(a) = g(d)

g(a) gets parsed successively as:

1. g(A0)
2. A1

and g(d) gets parsed successively as:

1. g(A4)
2. A1

Therefore, g(a) = g(d) .

6. INEQUALITY DETERMINATION

The previous discussion indicates how the question of whether two items are known to be equal is answered. However, the question of equality has two additional possible answers. Namely, the items may be unequal or no information as to their equality is known. Determination of the answer to these two questions is more complicated. In order to facilitate such work we also keep track of equivalence classes which are known to be unequal by use of a table known as ineqtable. This table is again accessed by left and right. Therefore, whenever a merge of two equivalence classes occurs, this table must also be updated. This means that between steps 4b and 4c of the equality determination algorithm given in fig. 4 an update is made of the inequivalent classes.

Two items may be shown to be unequal explicitly or implicitly whereas two items are equal only explicitly. The process of parsing allows the bypassing of special handling for implied equalities since if a sentence is not in the data base, then it is added to it while being parsed. This is what enables the recognition of f(a) = f(b) given a = b . Implied inequalities are also quite easy to detect. In this case two sentences are not explicitly known to be unequal (i.e., the equivalence classes containing them are not known to be unequal); however, when they are assumed to be equal, and thereby added to the data base, then a contradiction will occur. This contradiction is detected at the occurrence of a merge of two equivalence classes which are known to be unequal. Thus it is seen that the only modification needed to the equality determination algorithm is to check if the two about to be merged equivalence classes are known to be unequal. Moreover, if at any time during the implicit inequality phase any entries must be added to the data base, then the sentences in question can not be shown to be implicitly equal.

The revised algorithm for the addition of any equality to the data base as well as determining implicit inequalities is given in fig. 6. Note the use of maxneq to indicate the number of entries in ineqtable. The proof of the completeness of the inequality determination algorithm remains the same while the proof of the completeness of the inequality determination algorithm is similar to the former, and thus will not be repeated. Similarly, the algorithm can be encoded more efficiently in the style of fig. 5 or even as discussed in section 7.

```
To add an equality pair:

1.  classl:=result of parsing left half;
2.  classr:=result of parsing right half;
3.  mods:=nil;
4.  a.  m:=min(classl,classr);
    b.  n:=max(classl,classr);
    c.  for j:=1 step 1 until maxneq do begin
              comment:  update ineqtable and check for a contradiction;
          if right(ineqtable[j]) = n then right(ineqtable[j]):=m
          else if left(ineqtable[j]) = n then left(ineqtable[j]):=m;
          if left(ineqtable[j]) = right(ineqtable[j]) then "contradiction";
          end;
```

```
        d.  for j:=m+1 step 1 until maxclass do begin
                comment:  find candidate for implied equivalence;
              if null(right(eqtable[j])) then nil
                comment:  check if a deleted duplicate entry;
              else if atom(left(eqtable[j])) then nil
              else if member(m,cdr(left(eqtable[j]))) then mods:=merge(j,mods)
              else nil
              end;
        e.  for j:=n step 1 until maxclass do begin
                comment:  replace all instances of the higher numbered
                          equivalence class by the lower numbered one;
              if null(right(eqtable[j])) then nil
                comment:  check if a deleted duplicate entry;
              else begin
                if right(eqtable[j]) = n then right(eqtable[j]):=m;
                  comment:  change index to point to new head of equivalence
                            class;
                if atom(left(eqtable[j])) then nil
                else if member(n,cdr(left(eqtable[j]))) then begin
                    comment:  replace higher numbered equivalence class by the
                              lower numbered one;
                  subst(m,n,cdr(left(eqtable[j])));
                  mods:=merge(j,mods);
                    comment:  add to list of candidates for implied equivalence;
                  end
                else nil;
                end;
              end;
    5.  while not null(mods) do begin
            comment:  step through the list of candidate nodes and look for
                      implied equivalences;
          for k:=2 step 1 until length(mods) do begin
            if left(eqtable[mods[1]]) = left(eqtable[mods[k]]) then begin
                comment:  an entry appears more than once in the equality data
                          base;
              temp:=right(eqtable[mods[k]]);
              mods:=delete(k,mods);
                comment:  delete the higher numbered duplicate entry;
              right(eqtable[mods[k]]):=nil;
                comment:  check if two entries are already in the same
                          equivalence class;
              if right(eqtable[mods[1]]) NEQ temp then begin
                  comment:  two equivalence classes must be merged;
                class1:=temp;
                classr:=right(eqtable[mods[1]]);
                go to 4;
                end;
              end;
            end;
          mods:=cdr(mods);
          end;
```

Fig. 6 - Algorithm to Add an Equality Pair with a Check for a Contradiction

7. TIME AND STORAGE REQUIREMENTS

From a computational complexity standpoint, the equality determination algorithm is quite simple. Specifically, in parsing a sentence there are exactly as many reductions to be made as there are atoms and function names in the sentence. Moreover, the constant of proportionality is directly related to the size of the data base since the latter must be searched for the appropriate reduction. Of course, the search can be speeded up by keeping the data base sorted via a hashing function [Knuth73].

A more careful analysis enables us to obtain upper bounds for the storage required for the equality data base as well as obtaining average execution times for determining equality and updating the equality data base. Specifically, the maximum number of equivalence classes possible is when all of the axioms use different functions and atoms. In this case we would need at most one equivalence class per function and atom. Thus the upper bound on the number of equivalence classes is the total length of the axioms in the data base (say n). Each equivalence class requires one pointer to the head of its class (the right field of eqtable). The left field of each eqtable entry requires one location for the function name and one pointer for each argument. The maximum number of pointers and locations necessary for the left field of all entries in eqtable is 2n-1. This quantity is achieved when all atomic arguments are unique and thus there are no common subexpressions as well as no equality relations. In other words, this is the worst case for a single formula comprised of other formulas meeting these conditions. The truth of this claim is seen by noting that every atom requires two left field entries - one for its equivalence class and one for its occurrence as an argument in another formula; and every instance of a function name, but the outermost one, requires two left field entries - one for its occurrence in a formula and one for its formula's occurrence as an argument in another formula. For example, g(f(a)) requires one location for a, two locations for f(a), and two locations for g(f(a)) - i.e., 2*3-1=5 locations. Finally, since formulas are not of fixed length (i.e., they have a varying number of arguments), we need one marker pointer per eqtable entry to denote the length of its left field, or, depending on the method of implementation, to denote the fact that there are no more arguments. In fact, we need one such pointer per eqtable entry or at most n such pointers in total. Recalling that we need one pointer for each right field means that our equality data structure requires at most 4n-1 pointers in order to be able to handle a set of axioms containing a total of n atoms and function names. Actually, this upper bound is 4n-2m where m is the number of axioms present. Note the indistinguishability of pointers and locations.

Both the equality determination and equality data base updating algorithms need to do table lookup type operations. This is a factor which slows down these algorithms considerably. However, the problem can be alleviated by use of hash table methods for the equality determination algorithm, and by use of linked lists for the equality data base updating algorithm. For each equivalence class we will keep a linked list whose elements are all the eqtable entries in whose left field the equivalence class appears. Also a linked list is kept with each equivalence class for all the eqtable entries which are in the equivalence class (i.e., identical right field entries). This will add at most n pointers for hashing, n pointers for the left field links, n pointers for the right field links, and n pointers for the heads of the left field links chains. There is no need for a special pointer for the head of a right field links chain since the equivalence class is a member of the chain while in the case of the left field links chain the equivalence class is not a member of the chain. Thus the new maximum number of pointers is 8n-2m. However, it will generally be the case that two pointers can be stored per computer word and thus the storage requirement is 4n-m words. Actually, we need slightly more storage since we must also account for the hash buckets. However, this amount is rather insignificant in light of the rest of the required storage, and thus we can increase the number of hash buckets to reduce the probability of collisions in the table. It might also be desirable to have doubly linked hash lists to facilitate updating.

In order to clarify the proposed data structure we give a sample entry in fig. 7. Note that all pointers always refer to the first word of the entry. This data structure will reduce greatly the amount of work the updating algorithm must do because each of the links enables the rapid execution of one of the steps. The links between eqtable entries containing a certain equivalence class name as an argument and the links connecting all entries in the same equivalence class enable the rapid execution of steps 4d and 4e of fig. 6. While executing step 4e, wherever a substitution in the left field is made, the entry must be rehashed according to the new left field contents and entered in the appropriate hash chain. The latter eliminates the need for step 5 since the act of entering the item in the new hash chain also includes a check of its presence in the chain. In fact, we can now slightly modify the updating algorithm to keep a list of items having identical contents that have not yet been merged. Actually, this is the same as mods only now we know exactly which entries are identical where previously this was ascertained by means of step 5. No modifications are proposed to step 4c which checks if the proposed merge of classes will result in a contradiction. Moreover, we use a count pointer to indicate the number of arguments in the left(eqtable) entry (atoms are represented as functions of zero arguments). The count also acts as a filter when collisions occur in the hash table since equality checks will only need to be performed when both the hashed value and the number of arguments are identical.

Therefore, we have seen that our equality determination and updating algorithms can be implemented in a manner which is largely dependent on the speed with which items can be looked up in a hash table. Furthermore, the equality determination algorithm is seen to be almost linear, on the average, in the sense that the time necessary to decide if two strings are identical is proportional to the length of the two strings. Moreover, we have devised a data structure which allows the updating of the data base without performing useless operations (this includes the many passes over the data base that the algorithm in fig. 6 might possibly make). As a final observation, note that previous stipulations that all data base entries refer to lower numbered equivalence classes is no longer necessary. The reason this requirement was made was to enable us to avoid searching the entire table in steps 4d and 4e of the algorithm in fig. 6.

pointer to head of the equivalence class	pointer to next entry having the same hash value
first table entry having the equivalence class name as an argument	pointer to next table entry in the same equivalence class
function name or atom name	number of arguments
argument 1	pointer to next equivalence class in which argument 1 occurs as an argument
argument 2	pointer to next equivalence class in which argument 2 occurs as an argument
⋮	⋮
argument m	pointer to next equivalence class in which argument m occurs as an argument

Fig. 7 - Proposed Data Structure

8. CONCLUSION AND FUTURE WORK

We have succeeded in answering the original set of questions posed in section 1. Moreover, the order in which equality pairs are added to the data base has no bearing on the actual process of checking equality since each computation is in an equivalence class and at any time all possible equivalences are taken advantage of as shown in the algorithms and their proofs.

Some directions for future work include the ability to handle commutative and associative functions, transitivity of functions, and certain relations of equality for functions. In a LISP domain, this would include such examples as

CONS(A,B)=XCONS(B,A) , LESSP(A,B)=GREATERP(B,A) , CAR(CONS(A,B))=A , etc. These relations would be dealt with on an instance basis and not on a general variable basis. This means that when certain functions are encountered, equalities are added to the data base. Such a scheme could adequately deal with quite a large number of known relationships between functions. However, we still would not be able to deal with examples such as f(x)=x where x is a free variable (i.e., not an instance). When these extensions are made, the proof of the algorithm will have to be modified to read "equality of two items will be determined subject to its being derivable from the known set of equalities."

9. APPENDIX

Precedence relations for a grammar G=(V,T,P,S) are defined as follows (x, y, and z are arbitrary strings of length greater than or equal to zero over the vocabulary and Bi represents element i of the vocabulary):

Bi = Bj iff there exists k such that Bk ==> x Bi Bj y

Bi < Bj iff there exists k such that Bi = Bk and Bk *==> Bj x

Bi > Bj iff there exists k such that Bk *==> x Bi and Bk = Bj or there exists m,n such that Bm = Bn and Bm *==> y Bi and Bn *==> Bj z

A grammar is said to be simple precedence if:

(1) No two productions have the same right hand side.

(2) At most one of the three precedence relations =, <, and > hold between any two symbols of the vocabulary.

Theorem: The grammar G of fig. 2 is always simple precedence.

Proof: We first show that at most one precedence relation may hold between any two symbols of the vocabulary.

(1) a < b implies that b is a leftmost symbol of some production. Therefore b is an atom or "(". a = b implies that b appears adjacent to the right of a in some production. However, b is an atom or "(" neither of which can ever appear adjacent to the right of any symbol in a production. Therefore a < b and a = b is impossible.

(2) a > b implies that a is a rightmost symbol of some production. Therefore a is an atom or ")". a = b implies that a appears adjacent to the left of b in some production. However, a is an atom or ")" neither of which can ever appear adjacent to the left of any symbol in a production. Therefore a > b and a = b is impossible.

(3) a > b implies that a is a rightmost symbol of some production. Therefore a is an atom or ")". a < b implies that there exists a nonterminal symbol C such that a = C . However, this is impossible since no symbol can ever appear adjacent to the right of an atom or "(" in a production. Therefore a > b and a < b is impossible.

Thus we have shown that our grammar, G, always satisfies the criteria that at most one precedence relation ever holds between any two symbols of the vocabulary. Moreover, the updating algorithm preserves the uniqueness of right hand sides of productions, and therefore regardless of the additional equality pairs the equality grammar, G, is always simple precedence.

10. REFERENCES

[ChangLee73] - Chang, C., and Lee, R.C., Symbolic Logic and Mechanical Theorem Proving, Academic Press, New York, 1973.

[CockeSchwartz70] - Cocke, J., and Schwartz, J.T., Programming Languages and their Compilers, NYU Courant Institute, April 1970.

[DowneySethi77] - Downey, P.J., and Sethi, R., "Variations on the Common Subexpression Problem," Technical Report, Bell Laboratories, Murray Hill, New Jersey, 1977.

[GallerFisher64] - Galler, B.A., and Fisher, M.J., "An Improved Equivalence Algorithm," Communications of the ACM, May 1964, pp. 301-303.

[Knuth73] - Knuth, D.E., Sorting and Searching, Addison-Wesley, Reading, Massachusetts, 1973.

[London72] - London, R.L., "The Current State of Proving Programs Correct," Proceedings of the ACM 25th Annual Conference, 1972, pp. 39-46.

[Martin68] - Martin, D.F., "Boolean Matrix Methods for the Detection of Simple Precedence Grammars," Communications of the ACM, October 1968, pp. 685-687.

[McCarthy60] - McCarthy, J., "Recursive Functions of Symbolic Expressions and their Computation by Machine," Communications of the ACM, April 1960, pp. 184-195.

[Naur60] - Naur, P., (Ed.), "Revised Report on the Algorithmic Language ALGOL 60," Communications of the ACM, May 1960, pp. 299-314.

[Samet78] - Samet, H., "Proving the Correctness of Heuristically Optimized Code," to appear in Communications of the ACM (1978?).