EQUIVALENCE AND INEQUIVALENCE OF INSTANCES OF FORMULAS

by
Hanan Samet

Abstract

An algorithm is presented for determining whether or not two instances of formulas are equal based on previous equality and inequality declarations. The equality determination algorithm is shown to be linear along with a completeness proof.

I. INTRODUCTION

We are interested in a system to handle equality operations based on known equivalences and inequivalences of instances of s-expressions. Some of the requests which we would like to be able to respond to are:

1. Are two items known to be equal?
2. Are two items known to be unequal?
3. Is the equality of two items known?
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)?
8. The results of equality tests should be independent of the order in which the equality pairs are processed.

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

```
Ex. 1:      Given:      a = b
                        c = d
                        b = c
            Derive:     g(a) = g(d)

Ex. 2:      Given:      g(b) = f(a)
                        g(c) = f(b)
                        a = b
                        c = d
            Derive:     g(a) = g(d)

Ex. 3:      Given:      c = d
                        f(a) = a
                        a = c
            Derive:     f(f(a)) = a

Ex. 4:      Given:      f(b) = a
                        f(a) = a
                        f(f(a)) = c
            Derive:     f(f(b)) = c
```

Note that these examples, and all future examples, are written in the more familiar infix functional notation although our system is for s-expressions.

II. OTHER APPROACHES

One way of keeping track of the equivalences is by means of equivalence classes. When a new equality is seen, the current list of equivalences is updated to reflect all possible members based on the new equivalence. One pitfall of such an approach is that all possible equalities can not be generated since such a procedure will not terminate (i.e. 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. A specific case in point is when f(a,b)is known to be inequivalent to f(c,d). This implies that a≠b or c≠d or both, but not equality (i.e. a=b and c=d). This means that we don't want any contradictions.

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 indicates a process of equivalence by reduction which is like parsing. Thus, our problem perhaps can be specified in terms of grammars:

Given a grammar GS for our language of s-expressions 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 GS (henceforth referred to as L(GS)). The set of equalities can be considered as a set of symmetric productions where each member of the equality is interpreted to be a nonterminal symbol with an added production going from the nonterminal to its corresponding terminal symbol.

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

Given a pair of sentences of L(GS), 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, GE is a type 0 grammar whose decision problem is undecidable (decidable for context sensitive grammars but this is not such a grammar since for each production the length of the left hand side of the production is not necessarily ≤ the length of the right hand side of the production). In other words 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 transitivity information or of the syntax of L(GS). In other words transitivity must be rederived each time it is desired to check if two items are equal. 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.

III. SOLUTION

The procedure that we will use is based on some special properties of our grammar, GS, given below:

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

<atom> indicates a variable name
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.

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(GS) 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(GS) which has been encountered while processing equalities of L(GS). Moreover, the components of each valid sentence of L(GS) 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, GE, whose nonterminals 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. determine for each half of the equality 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 uniquely represented as f(<eq class name of a=b>)
and thus the two classes containing f(a) and f(b) are merged. In other words
all 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(GS)
is the same as parsing a sentence of of L(GS). The only difference is that
instead of making a reduction to the nonterminal S (and also the start symbol),
we reduce 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 .

Equality, in general, can not be determined for arbitrary grammars. However, in
our case GS has some special properties. The main problem in parsing is that the
sentence being parsed may not be in 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 GE is always simple precedence (see proof in section
VII) 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(GS)) 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(GS) . By atom we mean variable names and not function names.
Note that since equivalence classes always contain valid sentences of L(GS), 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 below: (note the use of <ecln> to
denote an equivalence class name)

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

```
(              =
)       ≥              ≥       ≥       ≥
```

In the ensuing discussion remember that we are always dealing with valid sentences of L(GS). The reason we cannot handle arbitrary grammars with our algorithm is that it is not usual to be able to identify when a reduction is to be made. This is
true even if we have a simple precedence grammar since if a handle can not be identified, then it is impossible to tell when a reduction is to be made. This rules out top-down parsing methods since they operate by finding reductions, and if such reductions do not exist, then we can't very well know when they ought to be created. The primary reason for our success is that our grammar, GS, has only one non-terminal symbol, namely the start symbol. Moreover, all equivalence classes are merely renames of the start symbol that allow us to keep track of what the start symbol represents. Thus reductions are easy to determine, and, once determined, the nonterminal to which the handle is being reduced is either found or created.

The equality data base consists of a set of entries each of which is a sentence of L(GE). Each entry is uniquely numbered 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:

right ==> left

Also 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 following algorithm no sentence is included in more than one equivalence class, and, in addition, each sentence appears only once in an equivalence class.

IV. EQUALITY DETERMINATION ALGORITHM

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
      if null right(eqtable(j)) then nil
      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
      if null right(eqtable(j)) then nil
      else begin
          if right(eqtable(j)) = n then right(eqtable(j))← m ;
```

```
          if atom left(eqtable(j)) then nil
          else if member(n,cdr left(eqtable(j))) then begin
            subst(m,n,cdr left(eqtable(j))) ;
            mods← merge(j,mods)
            end
          else nil
          end
      end
5.  while not null mods do begin
      for k← 2 step 1 until length(mods) do begin
          if left(eqtable(mods(1))) = left(eqtable(mods(k))) then begin
            temp← right(eqtable(mods(k))) ;
            mods← delete(k,mods) ;
            right(eqtable(mods(k)))← nil ;
            if right(eqtable(mods(1))) ≠ temp then begin
                classl← temp ;
                classr← right(eqtable(mods(1))) ;
                go to 4
                end
            end
          end ;
      mods← cdr mods
      end
```

In the above algorithm eqtable is an array, accessed by left and right, which contains one entry for each s-expression. mods is a sorted list containing pointers to entries in eqtable which refer to any members of merged equivalence classes. maxclass is the number associated with the last entry in eqtable.

The algorithm terminates since parsing (steps 1-3) 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. This insures that our grammar will always have the property that no two productions have the same right hand side.

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. 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). The reason step 5, for each element of mods, only checks subsequent entries of mods for its duplicate occurrence is that by the nature of the parsing algorithm each equivalence class can only be referred to by higher numbered equivalence classes.

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 modifying it whenever a reduction is encountered which has no corresponding nonterminal (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.

3. If the resulting equivalence class names from the parses are identical, 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 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:

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

ii. 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 GE which is true by virtue of GE being simple precedence. Therefore if two items are equal, then they will be in the same equivalence class.

Thus the above algorithm for determining equality is complete.

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.

V. EXAMPLES

The updating algorithm can be described as incorporating one of the approaches mentioned earlier as a solution to the problem. Parsing is the same as the process of substituting class names for all subexpressions known to be members of equivalence classes while step 5 is similar to substituting the newly derived equality everywhere it appears as a subexpression.

As an example of the updating algorithm, we will show how the data base is created and how an equality is determined for example 4.

1.	f(b)	yields	A0:	b	A0	returns A1	
			A1:	f(A0)	A1		
2.	a	yields	A2:	a	A2	returns	A2
3.	f(b)= a	yields	A0:	b	A0		
			A1:	f(A0)	A1		
			A2:	a	A1		
4.	f(a)	yields	A3:	f(A1)	A3	returns	A3
5.	a	yields	no change			returns	A1
6.	f(a) = a	yields	A0:	b	A0		
			A1:	f(A0)	A1		
			A2:	a	A1		
			A3:	f(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:	b	A0		
			A1:	f(A0)	A1		
			A2:	a	A1		
			A3:	f(A1)	A1		
			A4:	c	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 and thus f(f(b)) = c .

As a more complicated example we now examine example 2.

```
1.  g(b)     yields          A0:   b     A0    returns A1
                             A1:   g(A0) 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
and thus g(a) = g(d) .

VI. INEQUALITY DETERMINATION

Until now we have shown 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. This is done via a table of equivalence classes, ineqtable, which are known to be unequal. 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 an update is made of the inequivalent classes.

Two items may be shown to be unequal explicitly or implicitly wheras 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 manifested 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 eqquivalence 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 implicitly equal.

The revised algorithm for the addition of any equality to the data base as well as determining implicit inequalities is given below. 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 it will not be repeated.

```
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
          if left(ineqtable(j)) = n then left(ineqtable(j))← m
          else if right(ineqtable(j)) = n then right(ineqtable(j))← m ;
          if left(ineqtable(j)) = right(ineqtable(j)) then "contradiction"
          else nil
```

```
        end
    d.  for j← m+1 step 1 until maxclass do begin
      if null right(eqtable(j)) then nil
      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
      if null right(eqtable(j)) then nil
      else begin
          if right(eqtable(j)) = n then right(eqtable(j))← m ;
          if atom left(eqtable(j)) then nil
          else if member(n,cdr left(eqtable(j))) then begin
            subst(m,n,cdr left(eqtable(j))) ;
            mods← merge(j,mods)
            end
          else nil
          end
      end
5.  while not null mods do begin
      for k← 2 step 1 until length(mods) do begin
          if left(eqtable(mods(1))) = left(eqtable(mods(k))) then begin
            temp← right(eqtable(mods(k))) ;
            mods← delete(k,mods) ;
            right(eqtable(mods(k)))← nil ;
            if right(eqtable(mods(1))) ≠ temp then begin
                classl← temp ;
                classr← right(eqtable(mods(1))) ;
                go to 4
                end
            end
          end ;
      mods← cdr mods
      end
```

VII. CONCLUSION AND FUTURE WORK

At this point we mention that we have succeeded in answering the original set of questions posed in the introduction. 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 each point 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, transitvity of functions, and certain relations of equality for functions. This includes 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 handled 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 would still not be able to cope with examples such as f(x)=x where x is a free variable (i.e. not an instance).
In fact such a system will only treat equality between
functions, and not equality between functions and variables. 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".

VIII. PRECEDENCE

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

Bi = Bj iff ∃ k such that Bk →→ x Bi Bj y

Bi ≤ Bj iff ∃ k such that Bi = Bk and Bk *→→ Bj x

Bi ≥ Bj iff ∃ k such that Bk *→→ x Bi and Bk = Bj
or ∃ 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 part.

2. At most one of the three precedence relations =, ≤, and ≥ hold between any two symbols of the vocabulary

Theorem: GE 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.
However, b is an atom or "(" neither of which can ever appear adjacent to the right of any symbol.
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.
However, a is an atom or ")" neither of which can ever appear adjacent to the left of any symbol.
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 ∃ a nonterminal C such that a = C .
However, this is impossible since no symbol can ever appear adjacent to the right of an atom or "(".
Therefore a ≥ b and a ≤ b is impossible.

Thus we have shown that our grammar, GE, 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, GE, is always simple precedence.

10. REFERENCES

[1] McCarthy, J., LISP 1.5 Programmer's manual
[2] Allen, J., Interactive Theorem Prover
[3] Martin, D., Algorithms for Generation of Boolean Matrices for Computing Precedence Relations