

January 1971

A HEURISTICALLY GUIDED EQUALITY RULE
IN A RESOLUTION THEOREM PROVER

by

Claude R. Brice*
Paris, France

Jan A. Derksen
Artificial Intelligence Group
Stanford Research Institute
Menlo Park, California

*A paper presented at the International Conference on Artificial Intelligence, August 1969, London, U.K.

Artificial Intelligence Group
Technical Note 45R

SRI Project 8259

The research reported herein was sponsored by the Advanced Research Projects Agency and the National Aeronautics and Space Administration under Contract NAS12-2221.

*This paper was written while Mr. Brice was an International Fellow at Stanford Research Institute.

Abstract

A new way of handling the equality relation within the framework of a resolution theorem prover is described. The system uses a modification of Morris' E-resolution, a rule of inference to handle equality, controlled by heuristic tree search techniques. The modification makes possible an implementation to which new rules of inference may be added easily.

A Heuristically Guided Equality Rule
In a Resolution Theorem Prover

by

Claude Brice
Jan Derksen

I INTRODUCTION

The equality relation is widely used but difficult to axiomatize efficiently. We describe a new way of handling this relation within the framework of a resolution theorem prover. The system uses a modification of Morris' E-resolution, a rule of inference to handle equality, controlled by heuristic tree search techniques. The modification makes possible an implementation to which new rules of inference may be added easily.

Each time the resolution theorem prover makes an attempt to resolve two clauses, but cannot unify a pair of literals, the "equality tree" generator is called. A tree of clauses is built by substituting equal terms in one of the literals, until unification is possible. The growth of the equality tree is controlled by bounds on the processing time, the breadth and depth of the tree, and a "reluctance" function. The reluctance function associates a cost with each node of the tree and selects nodes for expanding with minimal cost. The function is a linear combination of several "features." Some of the features are the probability that the literals will unify, the length of the literals, the number of constants the literals have in common, and the length of the clauses in which the literals occur.

Section II gives information on terminology, theoretical background and completeness results for E-resolution and variants. Section III describes the heuristic machinery of the system. It includes also descriptions of the "equality tree" and the search strategies used to find a path from the root to the goal node of the tree. Four sample proofs are given in Section IV.

II E-RESOLUTION

It is assumed that the reader is familiar with the standard notation and terminology used in literature on resolution.* Among other methods for dealing with equality, paramodulation is relevant as E-resolution can be shown to be definable in terms of paramodulation with resolution^{1*}. Therefore, let us recall briefly the definition of paramodulation.

Paramodulation--This is a rule of inference that, given two clauses:

A

and

$$\alpha = \beta \vee B \quad (\text{or } \beta = \alpha \vee B) \quad ,$$

having no variables in common and such that A contains a term δ , with δ and α having a most-general unifier σ , forms A' by replacing in A_σ one single occurrence of δ_σ by β_σ and infers $A' \vee B_\sigma$.

Example: given the following two clauses:

$$c = d \vee \bar{Q}c$$

$$f(c) \neq f(d)$$

$$x = x \quad ,$$

letting A correspond with $f(c) \neq f(d)$, B with $\bar{Q}c$, $\alpha = \beta$ with $c = d$, and δ with

*References are listed at the end of this paper.

d in A' , then one can deduce by paramodulation the clause

$$f(c) \neq f(c) \vee \bar{Q}c$$

and, by resolution, the clause $\bar{Q}c$.

Thus, intuitively, paramodulation provides a way to make use of the substitutivity property of equality. The reflexivity property does not, in E-resolution, require special axioms, and if α and δ have no most-general unifier, the program tries to find one for β and δ (with the same definition for α , δ , and β as in the paramodulation definition). Each clause may generate many distinct paramodulators. One can define, following Ref. 1, the descendants of a clause C from a set S obtained by paramodulating into the literal l of C, in the following manner:

$$P^0(S, C, l) = \{C\},$$

$$P^1(S, C, l) = \text{the set of descendants obtained from C,}$$

and by induction:

$$P^k(S, C, l) = \text{the set of descendants obtained by paramodulating from the clauses of } P^{k-1}(S, C, l) \text{ into the literal } l.$$

A. E-Resolution Defined

Using the preliminary definition above, one can define E-resolution as follows:

Let $P^\infty(S, C, l)$ be the union of the $P^k(S, C, l)$ for the different values of k . C_3 is an E-resolvent of two clauses C_1 and C_2 iff there exist a descendant clause C_1' from $P^\infty(S, C_1, l_1)$ and a descendant clause

C'_2 from $P^\infty(S, C_2, l_2)$ such that C_3 is a resolvent of C'_1 and C'_2 and the literals resolved upon in C'_1 and C'_2 are those descended from l_1 and l_2 , respectively.

The intuitive idea behind these concepts is to be able to deal with the transitive property of equality. The two clauses C_1 and C_2 generate two trees of descendant clauses, and a path in one of these trees can be built by a chain of equalities and substitutions. This definition is of little help for programming purposes because of the necessity of developing two trees of paramodulants. It can be shown that a similar definition, but using only one tree, is equivalent. This definition is given below:

C_3 is an E-resolvent of two clauses C_1 and C_2 iff there exists C'_1 from $P^\infty(S, C_1, l_1)$ such that C_3 is a resolvent of C'_1 and C_2 and the literal resolved upon in C'_1 is l_1 or the descendant of l_1 .

This way of implementing E-resolution differs from the one used in Ref. 3, but is closer to the formal definition of E-resolution given by Anderson.¹

B. Completeness for Ground E-Resolution

To show that this modified definition of E-resolution is complete, we will show that it is complete for ground E-resolution and following Ref. 1, lift this result to the general level. Using the terminology of the first definition of E-resolution, let C_3 be a resolvent of the two clauses C'_1 and C'_2 . Let us denote C'_1 by $\{l'_1\} \cup L'_1$ and C'_2 by $\{l'_2\} \cup L'_2$, where L'_1 and L'_2 are the sets of literals from C'_1 and C'_2 not deleted by the resolution. C_3 is therefore expressed by $L'_1 \cup L'_2$

and, since we have performed ground resolution, l'_1 and l'_2 are complements. By applying in reverse order to the literal l'_1 of the clause C'_1 , the chain of equality replacements that took place to generate l'_2 , we can generate the clause $C''_1 \equiv \{l''_1\} \cup L''_1$, where $l''_1 = \sim l'_2$. C''_1 can then resolve against C_2 and as we apply the same chain of paramodulations, the resolvent is C_3 . It is shown in Ref. 1 that E-resolution, expressed here in terms of resolution and paramodulation, is complete with or without set of support, under the same condition as paramodulation: If the reflexivity axiom $(FA(x) \ x = x)$ (in which $FA(x)$ stands for $(\forall x)$) is present and all reflexivity functional axioms $(FA(x,y)(x = y \supset f(x) = f(y)))$ are also present.

C. Comments on E-Resolution

If E-resolution can be expressed in terms of paramodulation and resolution, one might wonder to what extent it is a useful technique. In contrast with paramodulation, E-resolution limits the number of clauses on which a theorem prover works. This is an important advantage because the strategies used to select clauses that are resolved upon are not very successful when the number of clauses increases. The reason is that in E-resolution a clause is added to the set of clauses of the system only if a resolvent is found, while in paramodulation the clauses that would be on the intermediate nodes of the E-resolution tree would be added. However, two clauses may yield more than one resolvent, and E-resolution is complete only if from two clauses one can generate all the possible distinct E-resolvents. To generate all the E-resolvents from two clauses C_1 and C_2 , one would have to grow from C_1 or C_2 a tree $P^\infty(S, C_1, l_1)$ or $P^\infty(S, C_2, l_2)$ until all the possible

distinct paramodulants of C_1 or C_2 into l_1 or l_2 have been found. In practice, these trees are almost never generated entirely. They could sometimes be infinite trees and, at any rate, the expansion of these trees is time consuming. Furthermore, in most cases not all the E-resolvents of two clauses are needed to obtain a proof. Instead, as in Ref. 3, one might use a tree-level bound on the depth of the E-resolution tree. This bound limits the number of nodes to be generated, and it is increased progressively until a proof is found. Of course, then some procedure to save the partially expanded trees of paramodulated clauses should be implemented. If a proof is not found, the tree-level bound is incremented and the search can continue using the saved trees. This tree-level bound serves also the purpose of limiting the search when it happens that two clauses have no E-resolvents. To further limit the search, Morris's E-resolution program³ was activated only if the two literals l_1 and l_2 of two clauses C_1 and C_2 did not unify. Although this limitation seems fairly natural, it made the system incomplete, as was shown by Anderson⁸ using the following set of clauses:

- (1) $P_{f(x)g(y)} \vee P_{h(x)i(y)}$
- (2) $\sim P_{f(a)g(b)}$
- (3) $\sim P_{h(b)i(b)}$
- (4) $f(a) = f(c)$.
- (5) $h(c) = h(b)$

The only possible resolvents obtained, if E-resolution is called only iff two literals do not unify, are:

$P_{h(a)i(b)}$

and

$P_{f(b)g(b)}$

However, by using the general E-resolution, one can generate the nil clause:

- (6) $Ph(c)i(b)$ E-resolvent from (1) and (2) obtained by paramodulation of (1) using (4)
- (7) nil E-resolvent from (6) and (3) obtained by paramodulation into (6) using (5)

D. The EQA3 E-Resolution System

The E-resolution program EQA3 was designed to be a package that could be added to QA3.² QA3 is a question-answering system based on the resolution principle, and it uses the set-of-support and unit-preference strategy. EQA3 does not differ in principle from the system described in Ref. 4 but actually suffers from its implementation in a theorem-proving system whose conception and structure do not lend themselves to improvement and refinement. However, QA3 will probably be rewritten or replaced by a more flexible theorem prover. With this purpose in mind, we have tried to keep EQA3 as general as possible. The descendant tree generator, which is the core of the system, could eventually be used to generate any type of clause that could be inferred using a rule other than paramodulation. For example, to express that the predicate $P(x,y,z)$ is such that the order in which its arguments appear is irrelevant, one can use the tree generator with a rule of inference consisting of a permutation of the arguments. Special permutations such as cyclic permutation can be used. For example, when the

orientation of a line is irrelevant, one wants to express that the two predicates $LINE(A,B)$ and $LINE(B,A)$ have the same truth value. A predicate may have only a few ground instances. A model for this predicate could then be the list of the ground instances for which the predicate is true. Each clause containing a model evaluable predicate can be resolved against or subsumed by a model unit clause.

One can imagine that the tree generator could select, depending on the problem, different rules of inference to produce descendant clauses. The selection can be simply as in E-resolution, e.g., the failure of rule 1 (resolution) implies the use of rule 2 (E-resolution). Another possibility is that each predicate has a property list that tells what rule of inference has to be used.

In its actual implementation, EQA3 makes use of an evaluation function or reluctance function to select the next node to be expanded. The system actually uses only paramodulation to infer new nodes in the tree. Not all the descendants of clauses obtained by paramodulation into a literal are generated. Some of them would be of no use in trying to get an E-resolvent. The substitution takes place only in the terms of the literal or descendant of the literal when the unification algorithm fails. Thus, this implementation suffers from the same incompleteness as Morris's system (see Section II-C), but this limitation has no practical consequences. The reflexivity axiom $X = X$ does not have to be added as an axiom to the set of clauses. It is "built into" the system and used to resolve against inequality literals.

Each clause containing an equality literal is collected on a list of equalities (EQLIST). This list is ordered by length of clause. Thus equalities belonging to shortest clauses will be tried first for possible application of the E-resolution rule. This was done in an effort to keep the length of the eventual E-resolvent to a minimum (cf. unit preference strategy).

The next section describes the structure of the tree generator and the reluctance function in more detail.

III THE EQUALITY TREE

The essence of E-resolution is that each time two literals with the same predicate but opposite sign do not unify, a special procedure is invoked to show these two terms to be equal using unknown equalities. The special procedure generates a tree of modified clauses obtained by substituting terms in the literal of the clauses until they unify, using a list of positive equalities. This tree is called the "equality tree."

The procedure is a rather general one given:

- (1) A start node, a string of symbols, e.g. $p(a)$.
- (2) A goal node, also a string of symbols, e.g. $p(b)$.
- (3) A list of equalities, e.g. $(a = c, c = b)$. We can replace part of a string by another string known to be its equal by a most-general common instance, and form in this way other nodes (grow a tree), e.g., we can form the nodes $p(c)$ and $p(b)$; the last node is equal to the goal node.

The procedure tries to find a path between the start and goal nodes. In our example the path found is $p(a), p(c), p(b)$. In our description

we showed only nodes consisting of single literals. In general, nodes have more complex values (clauses), although the search is done for specific literals in the start and goal clauses.

A. Search Procedures

Several difficulties arise if we want to implement such a procedure:

- (1) A path between two nodes does not have to exist; in this case we should be able to end the search.
- (2) Some branches of the tree must be recognized as unsuccessful, and the search stopped in favor of other branches.

Because of such problems, we cannot use a "depth-first" search technique as we are unable to recover from a single wrong decision in an infinite tree, e.g., we grow a branch from the node $f(x) = e * x$ and repeatedly apply the equality $x = e * x$. Then we get the branch

$$\begin{array}{c} f(x) = e * x \\ | \\ f(x) = e * e * x \\ | \\ f(x) = e * e * e * x \\ \vdots \end{array}$$

An alternative is a "breadth-first" search; we avoid the difficulty associated with infinite branches, but now we cannot use heuristic information in deciding which branch to develop first.

The method used in this system is search controlled by a reluctance function. In this method we may think of a "frontier" of those nodes whose successors have not been examined. This frontier is extended by choosing any node in it and examining the node's successors. The node chosen will be that which has the minimum value of some function called a "reluctance function." It is easy to see that breadth-first and depth-first search are special cases of this last search method.

B. How to Grow a Tree

Several factors have to be considered when growing a tree.

- (1) Control of the size of the tree; there are bounds on:
 - Breadth
 - Depth
 - Processing time spent working on this specific problem.
- (2) Successors or sons of the node. Not all of the sons of a node are developed at once because the number of sons may be unreasonably large, for instance, when a long list of equalities has to be used. Also, we may reach a goal node without having to find all the successors.
- (3) Some nodes are thrown away if associated cost values are above a "cutoff" criterion. In our system this is a limiting of the length of the literal relative to the length of the goal literal used. If the global variable LIMIT LENGTH is set to four, then any node with literals whose number of symbols (length) is four times the length of the goal literal will be thrown away.

- (4) Order of the evaluation of the nodes, the reluctance function: the essential function is EVALCOST.

The tree-growing program is written in an elegant recursive form as suggested by Burstall.⁵ The controlled search therefore could be programmed in a manner very similar to depth-first search.

C. The Reluctance Function

Each time a node is to be selected for expanding, the candidates are "open" nodes--nodes that do not have their maximum number of sons (limited by the breadth bound). The node selected is the one with minimal associated cost. The reluctance function should have a minimal value for the chosen node. The cost given by the reluctance function "EVALCOST" is a combination of several "features." A feature f_i is a function that measures some properties of a node. EVALCOST makes use of the linear combination

$$C = C_1 f_1 + C_2 f_2 \dots + C_n f_n .$$

Of course, nonlinear combinations are possible, but for our system the linear reluctance function is adequate.

D. Features Used in the Reluctance Function

The features are all functions of the string (literal) associated with the node under investigation. The goal literal is the string that is the ultimate goal of the tree search. The features are:

- (1) Length relative to the length of the goal literal
- (2) Number of constants in common with the goal literal
- (3) Number of function symbols in common with the goal literal

- (4) Degree of nesting relative to degree of nesting of the goal literal
- (5) Length of the clause of which the literal is an element
- (6) Probability that the literal and the goal literal will unify.

E. Detailed Description of the Features

- (1) Length: number of symbols in the string (literal) including parentheses, e.g., $(f(e)3) \rightarrow 7$.
- (2) Number of constants: A list of constants occurring in the literal is made and compared with the list of constants occurring in the goal literal.

An integer expressing the number of shared constants is computed. The algorithm gives, in this way, a maximal value for literals with an equal number of the same constants, while lower values are obtained for a smaller or larger shared number of constants.

- (3) Number of function symbols: each substring of the form ' $\langle \text{atom} \rangle \langle \text{atom or list} \rangle$ ' contains the occurrence of a function symbol, here, ' $\langle \text{atom} \rangle$ '. The procedure for the number of constants is reapplied, this time with the two lists of function symbols.

- (4) Degree of nesting: this feature is defined as $\sum K_x$, where x is an element of the string under consideration; x is not '('or)'; K_x is the number of bracket pairs enclosing x --e.g., '(g(f a)(e))' \rightarrow 7.
- (5) Length of clause: the literal associated with the node is an element of a clause. The resolution system has a built-in strategy with a preference for short clauses. In the E-resolution part, too, nodes with long clauses are penalized.
- (6) Probability that the literal and the goal literal will unify: normally if two literals do not unify, the unification algorithm fails and leaves us without information on how "well" the unification was doing. If the algorithm in this system fails, it reports the degree to which the literals did match. The algorithm adds
- For each term that matches $[1/\text{number of terms}]$, the weight of the term, to a running count
 - Inside a term, for each subterm, $[\text{weight of the term}/\text{number of subterms}]$ to the running count,
- The algorithm tries to unify the two literals and adds each time that the unification succeeds on a subterm the weight of the term or subterm of a term to the running count, e.g.
- (P(f x)(a)), (P(g x)(a)) \rightarrow 0.5
- (P(f x)(a)), (P(f y) z) \rightarrow 1
- (P(f(a))x), (P(f(b))(a)) \rightarrow 0.25 .

F. Performance of EVALCOST

Not much is known about the influence of the different features. The sample runs are made with a setting in which the length of the literal, and to a lesser degree the length of the clause of a node, determine the expansion of the tree. The user will have to experiment which setting to use for each problem. A semi-interactive use with tracing, printing and checking of the growth of the tree seems the best approach.

G. Structure of the Tree

1. Structure of the Tree

The tree is built with atoms generated by GENSYM. Each node has a property list with the following flags and information:

VALUE, the literal associated with the node

FATHER, a backpointer to the father node

SONS, a list of pointers to son nodes

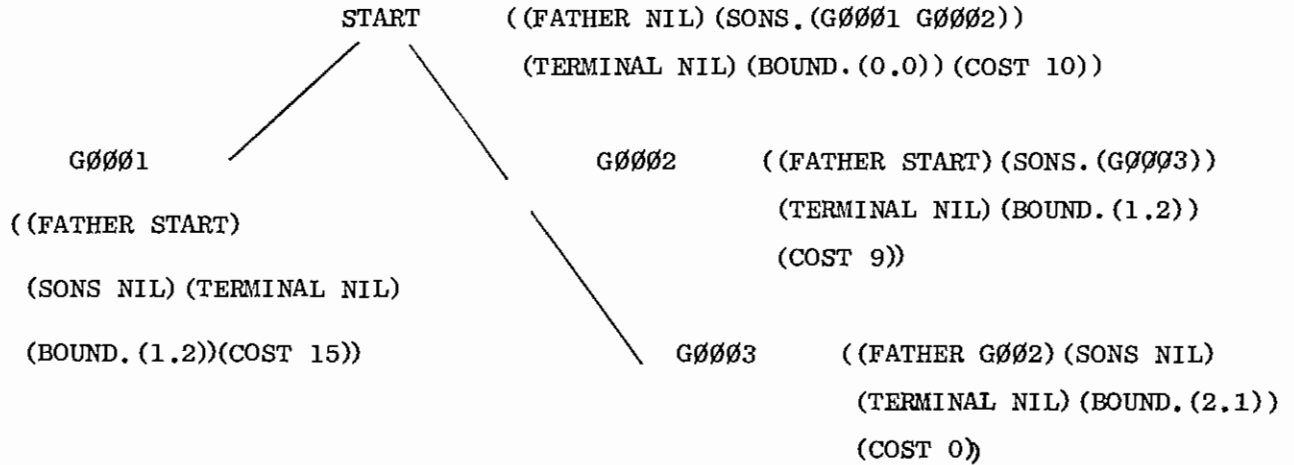
EXPBY, a pair of pointers to the terms of the literal and the equality, used in the substitution

TERMINAL, T if node is terminal, otherwise NIL

BOUND, dotted pair of current depth and breadth (number of sons) (in this order)

COST, cost as computed by EVALCOST.

2. An Example of a S^sall Tree



IV EXAMPLES OF RUNS

Three of the following examples come from elementary group theory.

'*' denotes the binary group operator. E denotes the identity of the group and INV denotes the inverse operator. One example is about the induction proof for the synthesis of the reverse function. REV denotes the reverse function, and AP, LIST, and CDR denote the LISP functions APPEND, LIST, and CDR. For each proof that implied only one E-resolution, the penetrance--the number of nodes developed on the path divided by the total number of nodes developed in the tree--is given. In the sample runs the theorems to be proven are preceded by TQ and the axioms by AX. Literals derived by E-resolution from a part of the negation of the theorem are preceded by NEG-THM.

Example 1 (See sample printout on page 18). The theorem proven is: the identity element of a commutative group is unique. The negation of the theorem is $(e1 * x = x) \& e1 \neq e$, or in prefix notation as used in the printouts: $= (*(E1,X)X) \& \neq (E1,E)$. EQA3 generates a contradiction by paramodulating into $\neq (E1,E)$ and resolving against the reflexivity axiom $= (X,X)$. A measure of goal directedness

of the proof is the ratio of the length of the path and the total number of nodes generated during the proof. This measure is called penetrance. The penetrance in example 1 was 3/8.

Example 2 In the proof shown on page 19 the group is restricted such that for all elements X, $(=(* X X)E)$. The theorem proven is $(\forall x) x \cdot x^{-1} = e$. Penetrance 8/22.

Example 3 Page 20 gives a sample printout for the proof of the theorem $(\forall x)(\forall y)(\forall z)x \cdot (y \cdot (x^{-1} \cdot y^{-1})) = e$. The penetrance was extremely low, 7/104. It seems that the main reason for such a bad performance was the great number of unproductive paramodulants that were generated by using the commutativity axiom $(FA(X,Y)(=(* X Y)(* Y X)))$.

Example 4 In the sample printout on page 21, SK45, SK46, and SK47 denote the constants that replace the universally quantified variables Y1, Y2, and X. As the system starts from the negation of the theorem to find a contradiction, the universal quantifiers of the theorem become existential quantifiers and the variables are replaced by Skolem constants. The instantiation of Y2*, not given by the system, is $Y2* = AP(LIST(CAR(SK45)),SK46))$ or for all Y1, Y2, and X the theorem is true with $Y2* = AP(LIST(CAR(Y1),Y2))$. Penetrance 3/5.

(QAS).
EXECUTIVE MODE

←RESET

←AX(AX1 AX2)

AX1 (FA (X Y) (= (* X Y) (* Y X)))

AXIOM

AX2 (FA (X) (= (* E X) X))

AXIOM

←TQ(IF(FA(X)(=(* E1 X) X))(= E1 E))

1. NEG-THM 0. =(* (E1, X), X)

2. NEG-THM 0. -= (E1, E)

3. NEG-THM 0. -= (* (E, E1), E)

CLAUSE 3. EQUAL.

4. NEG-THM 0. -= (* (E1, E), E)

CLAUSE 4. EQUAL.

5. NEG-THM 0. -= (E, E)

CLAUSE 5. EQUAL.

6. AXIOM 0. = (X, X)

7. RES(5. 6.) 1. CONTRADICTION

YES

Example 1

(QAS)
EXECUTIVE MODE

←RESET

←AX(AX2 AX3 AX4 AX5)

AX2 (FA (X) (= (* E X) X))
AXIØM

AX3 (FA (X) (= (* X X) E))
AXIØM

AX4 (FA (X) (= (* X (INV X)) E))
AXIØM

AX5 (FA (X Y Z) (= (* X (* Y Z)) (* (* X Y) Z)))
AXIØM

←TØ(FA(X)(= X (INV X)))

1. NEG-THM 0. == (SK 45, INV(SK 45))

2. NEG-THM 0. == (SK 45, *(E, INV(SK 45)))

CLAUSE 2. EQUAL.

3. NEG-THM 0. == (SK 45, (*(X, X), INV(SK 45)))

CLAUSE 3. EQUAL.

4. NEG-THM 0. == (SK 45, *(Y, *(Y, INV(SK 45))))

CLAUSE 4. EQUAL.

5. NEG-THM 0. == (SK 45, *(SK 45, E))

CLAUSE 5. EQUAL.

6. NEG-THM 0. == (SK 45, *(SK 45, *(X, X)))

CLAUSE 6. EQUAL.

7. NEG-THM 0. == (SK 45, (*(SK 45, Z), Z))

CLAUSE 7. EQUAL.

8. NEG-THM 0. == (SK 45, *(E, SK 45))

CLAUSE 8. EQUAL.

9. NEG-THM 0. == (SK 45, SK 45)

CLAUSE 9. EQUAL.

10. AXIØM 0. = (X, X)

11. RES(9. 10.) 1. CONTRADICTION

YES

Example 2

(GAS)
EXECUTIVE MODE

←RESET

←AX(AX1 AX2 AX4 AX5)

AX1 (FA (X Y) (= (* X Y) (* Y X)))
AXIOM

AX2 (FA (X) (= (* E X) X))
AXIOM

AX4 (FA (X) (= (* X (INV X)) E))
AXIOM

AX5 (FA (X Y Z) (= (* X (* Y Z)) (* (* X Y) Z)))
AXIOM

←T0(FA(X Y)(= E (* X(* Y(* (INV X)(INV Y))))))

1. NEG-THM 0. == (E, (* (SK 45, (* (SK 46, (* (INV (SK 45), INV (SK 46))))))

2. NEG-THM 0. == (E, ((* (SK 46, (* (INV (SK 45), INV (SK 46))), SK 45))

CLAUSE 2. EQUAL.

3. NEG-THM 0. == (E, ((* (SK 46, (* (INV (SK 46), INV (SK 45))), SK 45))

CLAUSE 3. EQUAL.

4. NEG-THM 0. == (E, ((* (* (SK 46, INV (SK 46)), INV (SK 45)), SK 45))

CLAUSE 4. EQUAL.

5. NEG-THM 0. == (E, ((* (E, INV (SK 45)), SK 45))

CLAUSE 5. EQUAL.

6. NEG-THM 0. == (E, (* (INV (SK 45), SK 45))

CLAUSE 6. EQUAL.

7. NEG-THM 0. == (E, (* (SK 45, INV (SK 45)))

CLAUSE 7. EQUAL.

8. NEG-THM 0. == (E, E)

CLAUSE 8. EQUAL.

9. AXIOM 0. = (X, X)

10. RES(8. 9.) 1. CONTRADICTION

YES

Example 3

(QAS)
EXECUTIVE MØDE

←RESET

←AX(AX2 AX3)

AX2 (FA (U V W) (= (AP U (AP V W)) (AP (AP U V) W)))
AXIØM

AX3 (FA (U) (IF (NØT (NULL U)) (= (REV U) (AP (REV (CDR U)) (LIST (C
AP U))))))
AXIØM

←TPRØVE Q1

(FA (Y1 Y2 X) (IF (AND (NØT (NULL Y1)) (= (AP (REV Y1) Y2) (REV X)))
(EX (Y2*) (= (AP (REV (CDR Y1)) Y2*) (REV X))))))

1. NEG-THM 0. -NULL(SK45)

2. NEG-THM 0. =(AP(REV(SK45), SK46), REV(SK47))

3. NEG-THM 0. -= (AP(REV(CDR(SK45)), Y2*), REV(SK47))

4. NEG-THM 0. -= (AP(REV(CDR(SK45)), Y2*), AP(REV(SK45), SK46))

CLAUSE 4. EQUAL.

5. NEG-THM 0. -= (AP(REV(CDR(SK45)), Y2*), AP(AP(REV(CDR(SK45)), L
IST(CAR(SK45))), SK46)) NULL(SK45)

CLAUSE 5. EQUAL.

6. NEG-THM 0. -= (AP(REV(CDR(SK45)), Y2*), AP(REV(CDR(SK45)), AP(L
IST(CAR(SK45)), SK46))) NULL(SK45)

CLAUSE 6. EQUAL.

7. AXIØM 0. =(X, X)

8. RES(6. 7.) 1. NULL(SK45)

9. RES(1. 8.) 2. CØNTRADICTION

YES

Example 4

V CONCLUSION

After some experimentation with our implementation of the equality rule several things became clear to us:

- (1) An equality rule without any guidance is doomed to fail on any nontrivial problem.
- (2) A heuristically guided search gives good results for a limited class of problems for which the reluctance function features are properly "tuned."
- (3) For a wider range of problems, watching the theorem prover in the process of proving an equality and introducing simple special rules of inference in case of failure or bad performance turns out to be the best solution.

REFERENCES

1. R. Anderson, "Completeness Results for E-Resolution," Proc. Spring Joint Computer Conference, pp. 653-656 (1970).
2. T. D. Garvey, and R. E. Kling, "User's Guide to QA3.5 Question-Answering System" Artificial Intelligence Group, Technical Note 15, Stanford Research Institute, Menlo Park, California (December 1969).
3. J. B. Morris, "E-Resolution: Extension of Resolution to Include the Equality Relation," Proc. International Joint Conference on Artificial Intelligence (Association for Computing Machinery, New York, New York, 1969).
4. C. C. Green, "The Application of Theorem Proving to Question-Answering Systems," (thesis) Artificial Intelligence Project Memo AI-96, Stanford University, Stanford, California, pp. 123-130 (June 1969).
5. R. M. Burstall, "Writing Search Algorithms in Functional Form," Machine Intelligence 3, D. Michie (ed.) (Edinburgh University Press, Edinburgh, Scotland, 1968).
6. R. J. Popplestone, "Beth-Tree Methods in Automatic Theorem Proving," Machine Intelligence 1, N. Collins and D. Michie (eds.) (American Elsevier Publishing Co., New York, New York, 1967).
7. G. A. Robinson, and L. Wos, "Paramodulation and Theorem Proving in First Order Theories with Equality," Machine Intelligence 4, B. Meltzer and D. Michie (eds.) (American Elsevier Publishing Company, New York, New York, 1969).