AISB

SUMMER CONFERENCE
JULY 1974

University of Sussex

CONTENTS

A NON-CLAUSAL THEOREM PROVING SYSTEM

by David Wilkins

ABSTRACT: There are reasons to suspect that non-clausal first-order logic expressions will provide a better base for a theorem prover than conventional clausal form. A complete inference system, QUEST, for the first-order predicate calculus using expressions in prenex form is presented. Comparison of this system with SL-resolution shows that clausal techniques can be transferred to prenex form and expected advantages do seem to appear.

KEY WORDS: resolution, clause, prenex form, SL-resolution

1- Introduction

A predicate calculus expression in prenex form is obtained from a given wff by eliminating implication signs, standardizing variables, reducing the scopes of negation signs, skolemizing existential quantifiers, and removing universal quantifiers; see Nilsson(1971) for a precise definition. Prenex form differs from conventional clausal form only in that distributivity is not repeatedly applied to yield an expression in conjunctive normal form. There are a number of reasons for suspecting that prenex form would be superior to clausal form in automatic theorem-proving.

As anyone who has converted large expressions to clausal form knows, the application of distributivity causes a multiplicative explosion in the number of literals so using prenex form will at least save storage and execution time (human or otherwise). Another advantage is that the same information is not spread over a number of clauses. If the expression $A \vee B \vee (C \wedge D \wedge E)$ is necessary in a refutation, a resolution-type system will resolve away A and B and one of C,D, or E. A clausal theorem prover may refute A and B and get stuck on C. It then has to back up and try clause ABD which will involve redoing the refutations of A and B. Current theorem proving systems do not avoid this redoing of work but this would be a natural result of using prenex form. With prenex form we also gain the ability to use subexpressions that are "anded" together. For example, in the expression $A \vee B \vee ((C \vee D) \wedge (\neg C \vee E))$ the $\neg C \vee E$ subexpression can be used to refute the $C \vee D$ subexpression without pulling in the "higher level" information that resolving against the clause $AB\neg C$ would. In this simple example, a clausal system could avoid re-refuting A and B but the ability to use subexpressions becomes valuable in the general case.

Theorem provers are considered inefficient problem solvers, but given an unsatisfiable set of predicate calculus expressions with no meaning attached to them, I would be worse than inefficient in finding a refutation. The theorem prover needs some

kind of knowledge about its input, or must at least be given advice. I will present a few reasons why I think prenex form is more suited for giving advice about than is clausal form. We find non-clausal forms easier to express ourselves in since we write axioms that way. Suppose I have an axiom which represents the fact F. If distributivity shatters this axiom into n clauses, the only plausible interpretation is that these clauses are all the possible cases that can occur. The advice changes from "use this axiom to prove F" to "here is a set of axioms related to F, use as many as are needed". Actually writing out expressions and looking at their clausal and non-clausal forms should convince the reader that clauses are not the best way to conceptualize things.

QUEST is a complete inference system for unsatisfiable sets of expressions in prenex form. It exhibits expected non-clausal advantages and is as computationally efficient as SL-reolution [4], one of the more sophisticated clausal inference systems.

2-Definitions

Prenex expressions are naturally tree-structured so I will use conventional terminology(Knuth 1968) to refer to trees except as noted below. Each tree has one particular node designated as the current node and this node is said to have control. Each node is the parent of the roots of its subtrees and each subtree is a son of the root node. Note that parent and son are not inverses. A node is an ancestor of a node, N, if and only if it is the parent of N or the parent of an ancestor of N. A tree is a descendant of a node N if and only if it is a son of N or a descendant of the root of a son of N. A node is active iff it is the current node or an ancestor of the current node. The cousins of a node, N, in a tree, T, are all those (and only those) subtrees of T that are sons of N or sons of an ancestor of N in T, and in addition are such that their root node is not an active node in T. A cousin of the current node in a tree is said to be a current cousin in the tree. Branch nodes will be either AND nodes or OR nodes while terminal nodes will be either **T**(true), **F**(false), or a literal.

If T is a tree and $\propto$ a substitution, then T*$\propto$ denotes the tree produced by applying $\propto$ to all nodes in T. Two trees, T1 and T2, are unifiable iff there is a substitution, $\propto$, such that T1*$\propto$ and T2*$\propto$ are isomorphic, in the sense that there is an isomorphism, f, from the nodes of T1*$\propto$ to the nodes of T2*$\propto$ such that if M and N are any two nodes in T1*$\propto$ the following three statements are true: 1) if N is an AND, OR, **T**, or **F** node then (N)f is the same type of node; 2)if N is a literal then (N)f is the same literal; 3)if M is the parent of N then (M)f is the parent of (N)f. T1 is said to be the same (sub)tree as T2 iff T1 and T2 are unifiable with the null substitution. The negation of a tree is formed by doing the following three things to the tree: 1)replace all **T** nodes by **F** nodes and vice versa, 2)replace all AND nodes by OR nodes and vice versa, 3)replace all literals by their negation.

I will now define the Truth Value Inference Rules which simply implement the definitions of "and" and "or". A node can be inferred false iff the node is an AND node and the root of one of its sons is **F**, or the node is an OR node and the roots of all its sons are **F**. A node can be inferred true iff the node is an AND node and the roots of all its sons are **T**, or the node is an OR node and the root of one of its sons is **T**.

The expression input to QUEST are assumed to be conjoined together, so an input set is represented by a tree whose root is an AND node and the subtrees of the root are the expressions in the input set.

An unsatisfiable tree, T, is minimally unsatisfiable iff when any subtree of T which is the son of an AND node and not the only son of that AND node is removed from the tree, the resulting tree is satisfiable.

3- Inference rules of QUEST

The truth value inference rules have already been mentioned. Let T be a tree from which we are trying to infer false. Suppose N is the current node in T and let S be the son of N we are currently trying to refute. When N is an OR node, all sons must be refuted but when N is an AND node the search strategy may pick a son to refute. To be complete, the inference system must in general allow any son to be tried although QUEST restricts the choice in some cases without sacrificing completeness. The distinction between rule of inference and operation must be understood. The rules of inference presented here are ways of changing a derivation tree so the validity of the expression it represents is unchanged. QUEST changes these rules into operations (of the same name) by allowing substitutions to be made in order to apply the rule and by placing restrictions on the use of the rule.

To develop the first rule, let O be the set of cousins of N in T which are sons of OR nodes, with S deleted from the set. Rule one considers all members of O to be false and infers whatever it can about S. Intuitively, the validity of this runs as follows. To obtain a refutation of T by working at N, all sons of OR nodes which are ancestors of N, i.e. O, must be inferred false if inferences about N are to help in a refutation. So if S is to be inferred false above N, it is safe to infer S false at N and wait for the refutation above N. This is valid because all inferences at a node are made using the information in a node's ancestors. Therefore, if S is the same subtree as a member of O, it is inferred false, and if it is the same as the negation of a member of O, it is inferred true. This rule is called the factoring rule of inference because it's role corresponds to the role of factoring in clausal inference systems. S is said to be factored on and the member of O is said to be factored against.

Let A be the set of cousins of N which are sons of AND nodes, with S deleted from the set. Rule two considers members of A true and infers whatever it can about S. This simply uses the information provided by the axioms used so far. If S is the same subtree as a member of A, it is inferred true, and if it is the same subtree an the negation of a member of A, it is inferred false. This rule is similar to ancestor resolution in linear resolution systems, but has added aspects because of the non-clausal structure. To avoid confusion, it will be called the smashing rule of inference. S is said to be smashed and the member of A is said to be smashed against. I shall call the combination of smashing and factoring the reduction rule of inference. Reduction and extension (a term used in the next paragraph) are both used in SL-resolution for similar ideas.

The last inference rule is the one corresponding most closely to resolution. It grows the current tree and is called the extension rule of inference. Extension says that members of A (the set defined in the last paragraph) can be grown onto N as follows: if N is an OR node then S can be replaced by a tree whose root is an AND node with its subtrees being S and a copy of a member of A; if N is an AND node then simply add a copy of a member of A as a new son of N. The member of A is said to be extended against and S is said to be extended on.

QUEST is basically these rules with strict restrictions put on them to guide the refutation and prune the search space.

4- Informal description of QUEST

QUEST has five operations. Each operation produces a new tree from an old tree. A QUEST derivation is a sequence of trees where each is produced by applying one of the five operations to its predecessor. The object is to produce the tree whose root is the node **F** from the input tree.

The most trivial operation is diving which simply moves control down one node to the root of a son of the current node. The truncation operation changes the current node to **F** and moves control up to its parent whenever the current node can be inferred false by a truth value inference rule. The reduction operation does inferences from the reduction rule of inference, but only when false is inferred, and then only when it is inferred from a son of the current node. The reduction operation, unlike the other two, may apply a substitution to the tree in order to make this inference. The deletion operation does true inferences from the reduction rule of inference, but only when a son of an AND node that is not the only son of that AND node is inferred true. Deletion may not apply a substitution since true inferences are a sign that something has gone wrong so we don't want to waste effort producing them.

The last operation is the extension operation. Growing the tree without a purpose will probably not get us any closer to a refutation so it will be required that the tree extended against have a subtree which is the negation of S. The number of subtrees in a tree increases with increasing tree size something like factorially or worse, so finding something to extend on may involve a huge number of unifiability tests. The test for unifiability of two trees is not trivial since the nodes at one level can be matched in many different ways with the nodes at that level in the other tree. Therefore QUEST does not allow extension on non-literal subtrees at all. Since only literals are extended on, the subtree extended against will always contain the negation of this literal after a substitution has been applied. This negation is called the literal extended against. The tree is grown as in the extension rule of inference and control is given to the root node of the extended against subtree. The extension operation, by definition, also requires that any AND node in the subtree extended against which is on the path to the literal extended against, must refute the son which contains the literal extended against. This does not destroy completeness and significantly prunes the search space since only one son need be tried at these AND nodes, and also makes the operation more like resolution since the extended against literal must eventually get smashed.

QUEST has six restrictions on the applications of these operations which significantly prune the search space.

1: The current tree can no longer be considered if an active node can be inferred true by the truth value inference rules and the reduction rule of inference. This does all true inferences not done by deletion. This restriction stops processing on trees that cannot lead to a proof.

2: The initial current node must be one that is in some minimally unsatisfiable subtree of the input tree. Thus only one starting point need be considered. This corresponds to support subset restrictions in clausal systems, but here the negation of the theorem can always be expressed in one prenex expression, so one starting point can be picked.

3: At a particular OR node, all reductions must be done before any extensions or dives. Without restrictions like this, the search space will be full of derivations that do the same operations in a different order. This eliminates all derivations which do extensions or dives before reductions at the same level. Hopefully, doing reductions first will instantiate the variables further thus reducing the number of possible unifications later on.

4: If any current cousin can be inferred true or false by the reduction rule of inference, then this is the only allowable operation. If there is a false inference the node should have been inferred false when its parent had control, but instead an extension or dive was done. Thus there is an easier proof than the one we are working on. If there is a true inference then either the first restriction will stop processing or a deletion will be done which simplifies the tree.

5: If the next operation is to be extension or diving then it must be done on the son selected by the selection function over OR nodes. This corresponds almost exactly to the selection restriction in SL-resolution. This also orders the applicable operations. If all nodes have n sons then a system without this restriction would have on the average n! times as many derivations it can produce. As would be expected, QUEST works for any selection function over OR nodes.

6: The same selection restriction is now applied to the sons that are reduced. The same factorial saving is made by not repeating the same reductions in different orders. The restriction is implemented by defining a total ordering of the sons rather than a function that picks one out. A function will not work because we cannot always apply reduction to a selected son. If the next operation is reduction then it can only be done if no son greater than (in the given ordering) the son being reduced has been reduced.

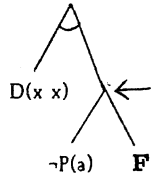
This gives an informal desription of QUEST that is exact as I could make it. The formal definition is fairly short and easy to read but is not within the scope of this paper. QUEST is sound and complete, but again the proofs are too long to present here. The formal definition and proofs can be found in (Wilkins 1973).

5- An Example

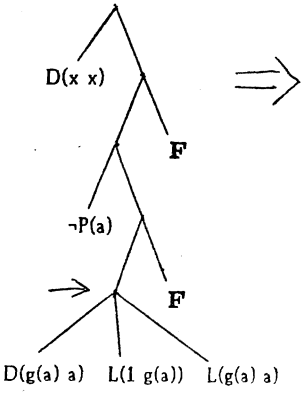
I will present one example of QUEST in action. I will point out places where prenex form is an advantage in the hope the reader will recognize these as general phenomena likely to occur in most problems. To make things readable, I will represent the trees graphically. AND nodes will be distinguished by drawing an arc through their branches. I will leave off the top AND node which has all the input expressions as its sons, but one should remember that it is there. The current node will be desgnated by an arrow.

Theorem: Every integer greater than 1 has a prime divisor. This can be axiomatized as follows: D(x x) means any number divides itself. ¬D(x y)∨¬D(y z)∨D(x z) represents the transitivity of divisibility. P(x)∨(D(g(x) x)∧L(1 g(x))∧L(g(x) x)) says that if x is not prime then a number between 1 and x divides x. Let a be the least counter-example to the theorem. The negation of the theorem is as follows: ¬P(x)∨¬D(x a) says that if x divides a then x is not prime. ¬L(1 x)∨¬L(x a)∨(P(f(x))∧D(f(x) x)) says that if x is between 1 and a then it has a prime divisor.

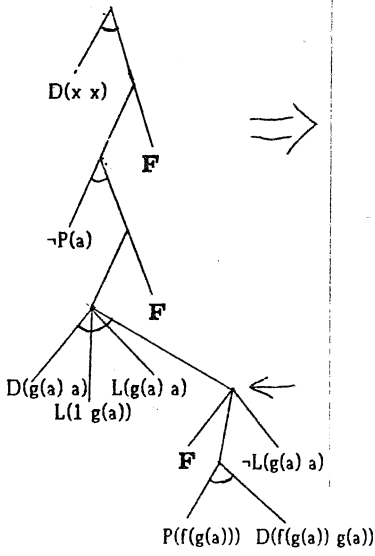
A proof found by a POP-2 program implementing QUEST is presented in the next five diagrams. The meaning easily attached to these diagrams is as follows: 1)Since a divides itself, it is not prime. 2)Thus there is a number, g, between 1 and a which divides a. 3)a was the least counter-example so there is a number, f, which is prime and divides g. 4)f does not divide a, since a is a counter-example. 5)By the transitivity of divisibility, this is a contradiction.



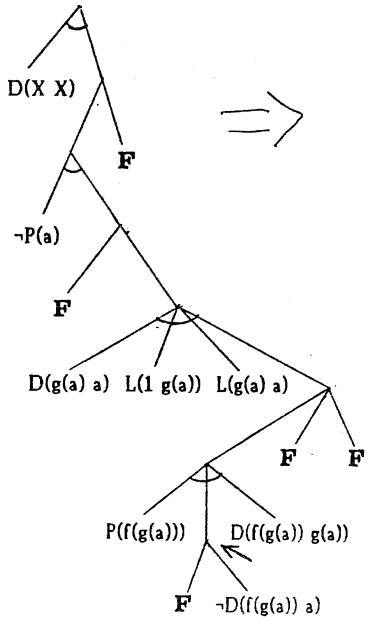
¬D(x a) is immediately smashed against input expression.



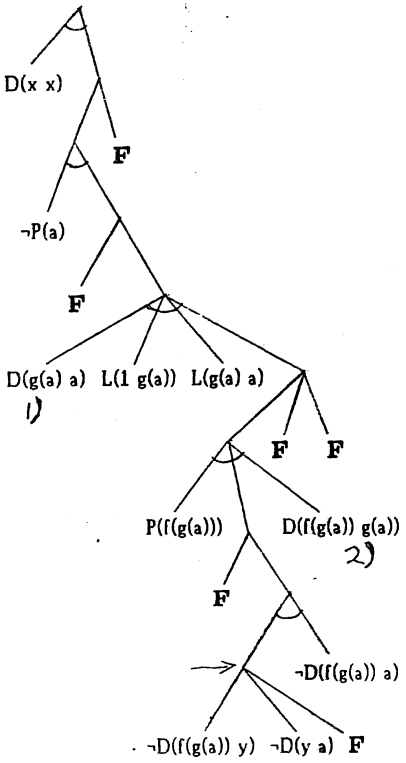
Extension on ¬P(a) against second axiom, followed by smashing the literal extended against. A literal must be chosen to extend on.



Extension on L(1 g(a)) against the fourth axiom, followed by smashing the literal extended against. The next operation must be the smash of ¬L(g(a) a) against the L(g(a) a). In clausal form, if we had just used the clause with L(1 g(a)) in it, we would need another whole clause to get L(g(a) a) and this would involve re-refuting all the literals above this AND node since distributivity would "attach" them to L(g(a) a).



Extension on P(f(g(a))) against the third axiom followed by smashing the literal extended against.



Extension against the first axiom with smashing of the literal extended against. Now y is instantiated to g(a) and both ¬Ds are smashed. Truth value inferences then infer **F** from the whole tree and the proof is complete. Note that the same situation as before arises when we smash the ¬Ds. They are smashed against 1) and 2), both of which are sons of AND nodes different from the son extended on at that AND node. Thus clausal form would have two more extensions against clauses rather than two reductions. I hope the reader may recognize this ability of prenex form as a general advantage and not particular to this problem. In a sense, extension in prenex form sucks in 2 or 3 or n clauses in compact form. Moreover they contain information likely to be relevant since one would usually not expect a single axiom to contain parts irrelevant to each other.

6-Comparison with SL-resolution

Many of the ideas in QUEST come from SL-resolution so it is natural to compare the two. For the reader not familiar with SL, it is presented in (Kowalski and Kuehner 1971). The purpose of this comparison is to show that our clausal techniques can be carried over to the prenex case, and to show how QUEST compares to clausal inference systems in general since SL is currently one of the better ones.

A QUEST derivation tree can be considered as an SL chain. A cousin which is the son of an OR node would be a B-literal, the son of an AND node would be an A-literal, and top to bottom would correspond to left to right. Let us consider the case when clausal input is given to QUEST. All cousins are now literals so I will speak of QUEST trees as if they were SL chains. The only difference between QUEST and SL chains initially is that QUEST chains have the unit clauses tacked on the front as A-literals. This was done because I felt extension against a unit clause is more like reduction than extension. Either system could easily be changed to be like the other. With clausal input, QUEST will never do a dive and will never do a deletion unless the same clause is input twice.

First, let us look at the admissibility restriction of SL. It says that no two literals in the chain may have the same atom unless the next operation is reduction. QUEST has the same restriction since two literals having the same atom is equivalent to being able to infer a cousin true or false by the reduction rule of inference. Let us now consider the operations.

The truncation operation is essentially the same in both systems. There are three differences in the reduction operation. Both systems require reductions at one level to be done before extensions but, as mentioned before, in QUEST this also applies to unit extensions. This is a trivial difference. SL does not allow factoring within a clause or ancestor resolution (smashing) against the rightmost A-literal while QUEST does. SL makes up for this by allowing extension against all factors of the input clauses. Thus SL has fewer reduction choices but more extension choices, but once again either system could easily be changed to be like the other. The third difference in reduction is the ordering of literals to determine the order of reductions. This simply applies the selection idea (the heart of the SL system) to reductions as well as extensions, and I feel it should be included in SL. The only difference in the extension operation is the already mentioned one of SL having more extension choices.

The differences when QUEST is applied to non-clausal input can be thought of as follows. Some links in the chain are now pointers to a tree instead of literals. These trees can be reduced as they are or "expanded in line" by diving operations. There are now A-links in the chain that correspond to sons of AND nodes in the input tree. These provide information which in the clausal case, loosely speaking, is only provided as a new clause and then only with more literals in the clause because of the distributivity applications. The following section gives evidence that the advantages one intuitively expects actually do appear.

7-An implementation

I wrote a POP-2 program implementing QUEST at the University of Essex. The program extends Boyer and Moore's structure sharing techniques (Boyer and Moore 1971) to the prenex case. The purpose of the program is to run on examples in clausal and prenex forms with a breadthfirst search so as to get a fair comparison of the size of the search space. Since QUEST is fairly good on clauses as shown by its comparison to SL, this should be a fair comparison. Three statistics are given: 1) cpu time in seconds, 2) number of extensions against input expressions, and 3) number of derivations being processed in parallel by the breadthfirst search.

For lack of space, I have picked only 3 examples. These demonstrate results found by running other examples. It was also found that the amount of processing needed in clausal cases varied greatly with the clause picked to start with. Problem 1 is the classical Quine-Wang problem P(x a)∨(P(x f(x))∧P(f(x) x)), ¬P(x a)∨¬P(x y)∨¬P(y x). Problem 2 is a variation of 1: G(y a)∨(G(y f(y))∧G(f(y) y)), ¬G(w y)∨(G(y f(y))∧G(f(y) y)∧¬G(y a)). Problem 3 is the example of section 5.

	NON-CLAUSAL			CLAUSAL		
	cpu time	exten.	deriv.	cpu time	exten.	deriv.
Problem 1	1.074	3	2	1.278	8	4
Problem 2	.746	3	2	1.717	11	7
Problem 3	11.296	65	45	28.965	136	74

8- Conclusion

It it may be favorable to abandon clausal form for a form easier to attach meaning to. This paper presents a non-clausal inference system that is complete at the general level and probably as efficient as current clausal systems. Section 7 provides evidence that computational advantages expected with prenex form do in fact appear (I do not wish to argue about judging criteria here). The comparison of QUEST with SL-resolution shows that techniques developed for clausal systems will be applicable to prenex systems.

REFERENCES

1. Boyer, R.S., and Moore, J.S., The Sharing of Structure in Resolution Programs, Metamathematics Unit, University of Edinburgh, 1971.

2. Hayes, P.J., and Kowalski, R.A., Lecture Notes on Automatic Theorem-proving, Metamathematics Unit Memo 40, University of Edinburgh, 1971.

3. Knuth, D.E., Fundamental Algorithms, Addison-Wesley, London, 1968.

4. Kowalski, R.A., and Kuehner, D.G., Linear Resolution with Selection Function, Artificial Intelligence, 2, 1971.

5. Nilsson, N.J., Problem Solving Methods in Artificial Intelligence, McGraw-Hill, New York 1971.

6. Wilkins, D.E., QUEST: A Non-Clausal Theorem Proving System, M.Sc. Thesis, University of Essex, 1973.