# THINKING

## *The Expanding Frontier*

Proceedings of the International, Interdisciplinary Conference on Thinking held at the University of The South Pacific, January, 1982

Edited by William Maxwell, *University of The South Pacific*
Preface by Jerome Bruner, *New School for Social Research*

Selected and Edited by
John Bishop, *University of Auckland*
Margaret Boden, *University of Sussex*
William Connell, *University of Sydney and Monash University*
Roderic Girle, *University of Queensland*
Jack Lochhead, *University of Massachusetts*
Jeanette Maas, *University of The South Pacific*
D. M. Mackay, *University of Keele*
S. Muralidhar, *University of The South Pacific*
David Perkins, *Harvard University*
R. A. C. Stewart, *University of The South Pacific*
Ivan Williams, *University of The South Pacific*

THE FRANKLIN INSTITUTE PRESS℠

Published by THE FRANKLIN INSTITUTE PRESS℠
Philadelphia, Pennsylvania



Current printing (last digit):
5 4 3 2 1

Printed in the United States of America

**Library of Congress Cataloging in Publication Data**
Main entry under title:

Thinking, the expanding frontier.

Proceedings of the Conference on Thinking, held in 1982 at the University of the South Pacific, Suva, Fiji under the auspices of the School of Education.
Includes bibliographies and index.
1. Thought and thinking—Congresses. 2. Cognition and culture—Congresses. 3. Bateson, Gregory—Congresses. I. Maxwell, William, 1935– . II. Conference on Thinking (1982 : Suva, Fiji) III. University of the South Pacific. School of Education.
BF455.T535 1983 153 83-1586
ISBN 0-89168-047-0

# HEURISTIC THEOREM PROVING[1]

Francis Jeffry Pelletier
Dan C. Wilson

## ABSTRACT

Many people view the essential difference between computer output and human thinking as "blindly following an algorithm versus heuristically using and altering strategies." A computer program which invokes heuristics is discussed and applied to the problem of producing proofs in ordinary logic. The conclusion is twofold: our computer program performs significantly better than existing algorithmetic methods, and it performs in a manner indistinguishable from the heuristics that humans use.

## THEOREM PROVING AND THINKING

In the last twenty years it has been common to mark off human performance from computer performance in terms of heuristics versus algorithms. A computer, so it is claimed, blindly follows a humanly devised algorithm and therefore cannot be counted rational, no matter what its performance. Humans, on the other hand, have a variety of heuristics or strategies at their disposal and engage in goal-directed thought by employing these strategies, so long as they appear to be leading toward the goal; they switch to another strategy when the previous one appears not to be succeeding. Now, the distinction between blind algorithms and heuristic strategies seems to us not to be very sharp and clear; we think that they merge into one another, and that the real difference between them is a subjective impression of unsureness of success in the case of heuristics. But we shall not argue that here. Instead, we shall accept the perceived distinction and display a program which uses exclusively what everyone would recognize as heuristic strategies.

The area where we intend to employ our heuristics is in theorem proving. We should perhaps indicate why we think this is a good and important testing place in

F.J. Pelletier is in the Department of Philosophy at the University of Alberta, Canada. D.C. Wilson is with A.G.S. Computers, Inc., in Mountainside, New Jersey.

1. We wish to thank Lenhart K. Schubert for his support of this project, both in terms of computer time and information. We also thank Jeffrey Sampson for encouraging us to develop our system in the first place. Dave Sharp supplied us with lists of "tricky" theorems to test THINKER with.

which to use heuristic techniques. The first reason is simply that logic has, since Aristotle, been considered one of the crucial areas that define rationality and thinking. Secondly, there are a variety of models of theorem proving (*i.e.*, systems of logic) available for use and their abstract properties are well-known. Thirdly, there are a variety of computer-based theorem provers around to which we may compare ours. Finally, and most importantly, the current state of artificial intelligence invokes theorem proving in a wide range of tasks which are taken to simulate thinking. For example, natural-language-understanding systems—including data-base retrieval and question-answering systems—all require a theorem prover to go from "what is literally said" to "what is meant by the speaker."

It is well known that to understand a normal speaker, a lot of "inferencing" is required to take into account background information, mutually known intentions, and what is likely to be meant in the current situation. The currently believed best way to handle this is to have a theorem prover take what is literally said, add it to this knowledge base, and construct likely conclusions about what was meant on this occasion. Also, current models of planning and action use theorem provers. Thus a robot is ordered to move Box A to Location Z. To do so, our robot first tries to prove that A is already at Z. When this proof fails, it inspects the proof to determine the simplest position, Y, it could be in which (together with one of its primitive movements) will allow it to prove that A is at Z. It then tries to prove that it is in position Y. When this proof fails, it inspects the new proof to determine what is required to prove Y. This is continued until it can prove that some series of primitive movements will get A to Z. And then it performs that series of movements.

The claim in artificial intelligence is that some similar procedure occurs when humans are planning actions and understanding language. Thus it would seem to be of paramount importance for research into thinking to be able to have a theorem prover which operates in a manner akin to human theorem proving. And, as we indicated earlier, we think that heuristic-based theorem proving provides the most likely hope in this regard.

The subject of what theorems can be proved in classical logic was pretty thoroughly canvassed by Whitehead and Russell in 1910–1912; and while various new and interesting theorems were discovered in the decades since, the truly exciting work in logic has been done at the model-theoretic level, and not *within* the logic itself. We therefore wish to emphasize that the present study of theorem proving is done with an eye toward discovering how people in general, when working with abstract matters of logic, actually proceed in formulating a proof (and not an investigation hoping to prove new theorems). We have another eye toward implementing such a theorem prover within natural-language-understanding systems and within robotic-planning systems. In this regard, Feigenbaum and Feldman (1963, p. 107) say:

> The fascination with mechanical theorem proving . . . lies less with the end (the production of theorems, perhaps new and important) than with the means (a thorough understanding of the organization of information processing activity in mathematical discovery). It is felt that understanding these problem-solving processes is an important step toward the programming of more complex, more general problem-solving processes for a variety of intellectual tasks.

## SYSTEMS OF LOGIC

In the abstract, systems of logic can be divided into three sorts: axiomatic, "semantic," and natural deduction. We wish to justify our choice of natural deduc-

tion as appropriate, especially in light of the facts that the earliest theorem prover was based on an axiomatic system and that almost all present theorem provers are "semantic" in nature. To this end, we shall describe the three types in this section, and in the next section point out what we take to be flaws in the first two types when it comes to computer theorem proving.

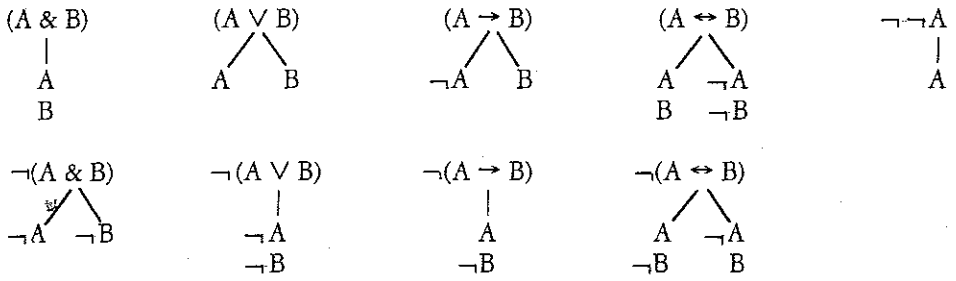
An axiomatic system of logic takes certain formulae as "given" (in the sense of requiring no other justification), gives a set of "rules of inference" (methods of transforming one or more formulae into another), and defines a *proof* as an ordered (finite) set of formulae, each one of which is an axiom or follows from previous (in the ordering) formulae by a rule of inference.

To give an example of a proof in a typical propositional axiomatic system, consider the system P1 of Church (1956). Included in the axioms are A1: $(p\rightarrow(q\rightarrow p))$, A2: $((s\rightarrow(p\rightarrow q))\rightarrow((s\rightarrow p)\rightarrow(s\rightarrow q)))$; and the two rules of inference, MP: from $(A\rightarrow B)$ and A, infer B; and Sub: from A, if $b$ is a propositional variable in A, infer the result of replacing all occurrences of $b$ in A by a formula B. A proof of the theorem $(p\rightarrow p)$ in this system would be

| | | |
|---|---|---|
| 1. | $((s\rightarrow(p\rightarrow q))\rightarrow((s\rightarrow p)\rightarrow(s\rightarrow q)))$ | A2 |
| 2. | $((s\rightarrow(r\rightarrow q))\rightarrow((s\rightarrow r)\rightarrow(s\rightarrow q)))$ | 1, Sub (r for p) |
| 3. | $((s\rightarrow(r\rightarrow p))\rightarrow((s\rightarrow r)\rightarrow(s\rightarrow p)))$ | 2, Sub (p for q) |
| 4. | $((p\rightarrow(r\rightarrow p))\rightarrow((p\rightarrow r)\rightarrow(p\rightarrow p)))$ | 3, Sub (p for s) |
| 5. | $((p\rightarrow(q\rightarrow p))\rightarrow((p\rightarrow q)\rightarrow(p\rightarrow p)))$ | 4, Sub (q for r) |
| 6. | $(p\rightarrow(q\rightarrow p))$ | A1 |
| 7. | $((p\rightarrow q)\rightarrow(p\rightarrow p))$ | 5, 6MP |
| 8. | $((p\rightarrow(q\rightarrow p))\rightarrow(p\rightarrow p))$ | 7, Sub ((q→p) for q) |
| 9. | $(p\rightarrow p)$ | 6, 8MP |

The extension of the axiomatic method to the predicate calculus is accomplished by adding further axioms and rules of inference.

"Semantic" systems of logic are so-called because they attempt to mirror the intended semantical interpretation into the system of logic itself. For the propositional logic, this intended semantical interpretation is just the truth table, and consequently, to prove whether a formula A is a theorem or not, it is customary in these systems to introduce devices which enable us to find out whether the assumption that A is *not* a theorem would also require that some atomic sentence and its negation both be assigned True. This is normally done by "breaking down" the formula ¬A into simpler and simpler components. Many of these methods can easily be represented by trees. Jeffrey (1967) has the following system of rules for tree construction:

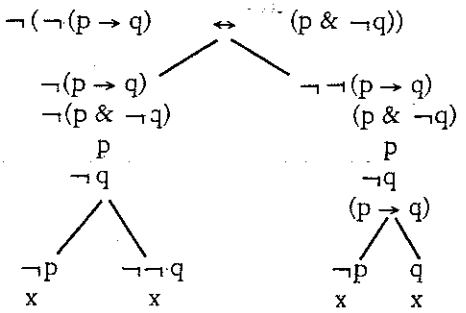



where the intuitive idea behind a rule is that we are interested in "ways the complex formula might be true." The definition of a proof of conclusion C is: *1.* ¬C is the

root node, *2.* if a rule of inference is applied to a formula B which occupies a node of the tree, the result of the rule of inference is represented in every "uncancelled" branch that B dominates, *3.* any branch which contains an atomic formula and also its negation is "cancelled" by putting an "x" at the bottom of the branch, *4.* C is a theorem if and only if every branch is cancelled. The proof of the formula $(p \rightarrow p)$ is very simple in this system. We put $\neg(p \rightarrow p)$ as the root node and use the rule for $\neg(A \rightarrow B)$:

$$\begin{array}{c} \neg\,(p \rightarrow p) \\ | \\ p \\ \neg\, p \\ x \end{array}$$

A somewhat more interesting theorem is DeMorgan's $(\neg\,(p \rightarrow q) \leftrightarrow (p\ \&\ \neg\, q))$, which is proved:




It is quite clear here (as opposed to the axiomatic system) what the strategy is: we assume the (alleged) theorem to be false, and break it down into simpler and simpler components by the truth-preserving rules (the branched formula is true if, and only if, at least one of its sub-branches is true). Since the resulting formulae get shorter and shorter, the method (in the propositional calculus) is guaranteed to halt.

In computerized theorem proving, the most commonly used method is "resolution"—a variant of the semantic methods. Here (in the propositional logic case) we negate the formula to be proved and represent it by its equivalent "clause form" in which the formula is converted to a conjunction of disjunctions of "literals" (=atomic formulae or their negations), and each conjunct is called a *clause*. Each clause (which is itself in disjunctive normal form) is written on a separate line and the one rule of inference, "resolution," is used. The rule is (in its simplest statement):

$$\begin{array}{l} A1 \vee B1 \vee \ldots \vee P1 \vee \ldots \vee Z1 \\ A2 \vee B2 \vee \ldots \vee \neg P1 \vee \ldots \vee Z2 \\ \hline \therefore\ A1 \vee B1 \vee \ldots \vee Z1 \vee A2 \vee B2 \vee \ldots \vee Z2 \end{array}$$

where: each of the lines is in clause form and the conclusion (a new clause) has no mention of P1 or its negation (it has been "resolved out"). If the original formula was a theorem, then eventually the method will yield a *null resolvent*—the "empty formula," a formula with no subformulae. (The rule is usually generalized to apply

to an arbitrary number of premises at one swoop.) Clearly the resolution method is semantic in nature: we are trying to discover whether the purported statement is necessarily true by looking at ways its negation might be true. If none are found (null resolvent), the negation can't be true and so the original statement must be.

These semantic methods can be extended to the (non-decidable) predicate calculus in various ways. Jeffrey (1967) adds branching rules for quantifiers, but these rules are not effective in the sense that they needn't ever be used again. Another way would be to convert the formula into a Skolem normal form by *1.* getting the prenex normal form (all quantifiers have widest scope); *2.* having every variable bound by an existential quantifier which is *a.* not in the scope of a universal quantifier is replaced by a name, *b.* in the scope of a universal quantifier is replaced by a (Skolem) function of the variable(s) mentioned by the universal quantifier(s); and *3.* dropping universal quantifiers. The resulting (non-quantified) formula can now be treated in various ways. We could apply the tree method of above, or we could continue to use the resolution procedure by introducing a special understanding of what variables can resolve against which and generate the null resolvent. Even in the complex case of quantifiers, it should be clear that the strategy is *semantic*: quantifiers are interpreted—existential quantifiers not in the scope of a universal are replaced by a name (the thing in the model that the sentence asserts the existence of), existential quantifiers in the scope of a universal quantifier are replaced by a function of the things named in the model by the universal quantifiers, and so on. Finally, we merely look to the possible co-truth of the atomic formulae.

A natural deduction system is like the semantic systems and unlike the axiomatic systems both because it has no unjustified statements (axioms) and because it has a large number of rules of inference; however, it is unlike the semantic systems in that it does not attempt to "break formulae down" into simple components and evaluate their possible co-truth. Rather, the rules of inference are supposed to correspond to psychologically plausible modes of reasoning. A proof is a method of breaking down a formula into "what you can assume" and "what still needs to be proved," together with methods to actually do some of the "proving." There are a number of these natural deduction systems in the literature; we shall here present (and later employ) the one found in Kalish & Montague (1964). For the propositional logic, the *rules of inference* are:

$$\frac{A}{A}\ (R);\quad \frac{A}{\neg\neg A}\ \text{and}\ \frac{\neg\neg A}{A}\ (DN);\quad \frac{(A\ \&\ B)}{A}\ \text{and}\ \frac{(A\ \&\ B)}{B}\ (S);$$

$$\frac{A}{(A \vee B)}\ \text{and}\ \frac{A}{(B \vee A)}\ (Add)\qquad \frac{(A \rightarrow B)\quad A}{B}\ (MP);\qquad \frac{(A \rightarrow B)\quad \neg B}{\neg A}\ (MT);\qquad \frac{A\quad B}{(A\ \&\ B)}\ (Adj);$$

$$\frac{(A \vee B)\quad \neg A}{B}\ \text{and}\ \frac{(A \vee B)\quad \neg B}{A}\ (MTP);\qquad \frac{(A \rightarrow B)\quad (B \rightarrow A)}{(B \leftrightarrow A)}\ (CB);\qquad \frac{(A \leftrightarrow B)}{(A \rightarrow B)}\ \text{and}\ \frac{(A \leftrightarrow B)}{(B \rightarrow A)}\ (BC)$$

which abbreviations stand for, respectively R: Repetition, DN: Double Negation, MP: Modus Ponens, MT: Modus Tollens, S: Simplification, ADj: Adjunction, Add: Addition, MTP: Modul Tollendo Ponens, BC: Biconditional to Conditional, CB: Conditionals to Biconditional. These are all taken to be psychologically plausible modes of reasoning. An *antecedent line* is defined as a line which is earlier in

the proof and neither boxed nor containing an uncancelled "show" (both defined below). A *proof* is defined as:

1. If A is a formula, then "Show A" can occur as a line. (The "show" is *uncancelled*. Intuitively we are setting the task of proving A.)

2a. If "Show A" occurs as a line then ¬A can occur as the next line ("assume the negation").

2b. If "Show ¬A" occurs as a line, then A can occur as the next line.

2c. If "Show (A→B)" occurs as a line, then A can occur as the next line ("assume the antecedent").

3. If C follows from antecedent lines by a rule of inference, then C may be entered as the next line.

4. If the proof has a subpart which looks like

Show A
X1
.
.
.
Xn

and *a.* there are no uncancelled "Show" among X1...Xn, and *b.* either A occurs unboxed (defined below) among X1...Xn, or else both C and ¬C occur unboxed among X1...Xn, then

Show A
X1
.
.
.
Xn

can be the next step in the proof (X1...Xn are now *boxed* — and thus are no longer antecedent, and the "Show" line is *cancelled* and now antecedent (intuitively, the lines in the box constitute a proof of A).

5. If the proof has a subpart which looks like

Show (A→B)
X1
.
.
.
Xn

and *a.* there are no uncancelled "Show" among X1...Xn, and *b.* B occurs unboxed among X1...Xn, then

Show (A→B)
X1
.
.
.
Xn

may occur as the next step to the proof.

6. The formula A is proved if it occurs unboxed in a proof and there are no uncancelled "Show" in the proof. (The method can be extended to arguments with premises by allowing a premise to be entered anywhere in the proof.)

This natural-deduction system is extended to the predicate calculus by adding rules for Existential Instantiation, Universal Instantiation, Existential Generalization, and another method of boxing and cancelling called universal derivation. The system we shall exhibit below is of the full predicate calculus, and hence uses these other quantifier rules also. We close this section with two short proofs to give a feeling for how theorems might be proved using this system. First, the theorem $((p \rightarrow q) \rightarrow (\neg q \rightarrow \neg p))$, which was the longest proof completed by the Logic Theorist (to be described below).

| | | |
|---|---|---|
| 1. | Show $((p \rightarrow q) \rightarrow (\neg q \rightarrow \neg p))$ | |
| 2. | $(p \rightarrow q)$ | Assumption |
| 3. | Show $(\neg q \rightarrow \neg p)$ | |
| 4. | $\neg q$ | Assumption |
| 5. | $\neg p$ | 2,4MT |

Second, the theorem $(p \vee \neg \neg \neg p)$ of which it has been proved that the Logic Theorist cannot prove it.

| | | |
|---|---|---|
| 1. | Show $(p \vee \neg \neg \neg p)$ | |
| 2. | $\neg(p \vee \neg \neg \neg p)$ | Assumption |
| 3. | Show $\neg p$ | |
| 4. | $p$ | Assumption |
| 5. | $(p \vee \neg \neg \neg p)$ | 4,Add |
| 6. | $\neg (p \vee \neg \neg \neg p)$ | 2,R |
| 7. | $\neg \neg \neg p$ | 3,DN |
| 8. | $(p \vee \neg \neg \neg p)$ | 7,Add |

## SOME REMARKS ABOUT PREVIOUS THEOREM PROVERS

The first theorem prover was the Logic Theorist of Newell, Simon, and Shaw (1957; its latest incarnation is in Newell and Simon, 1972). It employed the axiomatic system of Whitehead and Russell and was heuristic-based in the sense that it made use of such procedures as "Is there an axiom, the consequent of which is a substitution instance of what we're trying to prove? Yes — try to prove the axiom's antecedent;" ("backward chaining"), or "Are any of our current lines a substitution instance of the antecedent of an axiom? Yes — use MP"; ("forward chaining"), and so on. However, the Logic Theorist was not very successful in proving theorems. We have already indicated above that it could not prove $(p \vee \neg \neg \neg p)$ and that, in fact, the longest theorem it could prove was the simple $((p \rightarrow q) \rightarrow (\neg q \rightarrow \neg p))$. Furthermore, its proofs, of the few it could prove, tended to be extraordinarily inelegant. Part of the problem was the machine being used—it had limited storage and these strategies required a very large amount of storage of possible "substitution instances" of axioms and previously proved lines. We shall show below how we have solved this storage problem.

Another part of the problem was that the strategies employed were just too simplistic to be taken seriously. But another, more important, part of the problem is that the Logic Theorist used an *axiomatic* system. Almost every person (professional logician or student) finds axiomatic systems very difficult. If we want to mirror actual logical abilities, we would do better to look at how we learn one of the other versions of logic. Therefore, both on the grounds of its technical difficulties and the grounds of its implausibility as a model of thinking, we reject axiomatic systems.

We also reject the semantic systems. We have three reasons for this. The first is the abstract, theoretical consideration that semantics just isn't really logic, and it is

people's logical abilities we are concerned with. On this issue we quote from Georgacarakos and Smith (1978: xiv):

> In keeping with our aim of theoretical soundness, we have sharply distinguished between the semantical and the syntactical correlates of the logical concepts we study throughout the text. We introduce the technique of tree construction as a semantical device, the aim of which is to discover counterinterpretations for invalid argument forms. Many authors regard trees as syntactical devices and of course in a sense they are (they involve manipulation of symbols). However, the correct purpose of tree construction is semantical in that it is to be used as a device to find possible counterinterpretations.

The second reason is that the use of the semantical logics lends itself too easily to methods which are beyond the ken of reasoning people (*viz.*, resolution procedures). We wish to stick strictly to what ordinary people can actually know and use. It is the unanimous voice of resolution theorists that their techniques are too complex and time consuming for people to use. (Many writers claim this, but see Chang and Lee 1973.) The third reason we reject semantic methods is technical. The method used is, when not augmented by any heuristic search control, very inefficient, time consuming, and storage consuming. We quote here from Kowalski (1979, p. 163):

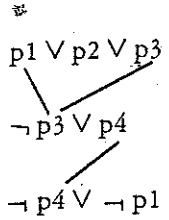
> The search space determined by unrestricted application of the resolution rule is highly redundant. Redundancy can be avoided, at the cost of flexibility, by restricting resolution to top-down or bottom up inference.

One might have to live with this unhappy state of affairs presented by these semantic, resolution provers if there weren't any better method. One way to make the method better is to control the search by certain heuristics. The mechanical theorem-proving literature is rife with suggestions, such as "set of support," "unit preference," "purity," *etc.*, but it must be admitted that these improvements in performance are at the expense of even what tenuous link resolution provers may have to human theorem proving. Two of these techniques ought to be mentioned here, nonetheless, since it is our aim to show that our program performs better than the best resolution prover (which uses one of these methods), and it does so because it more fully implements some of the ideas and techniques in the other.

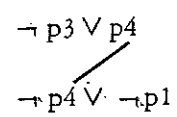
Kowalski (1974, 1979) is the most fully described "connection-graph" theorem prover. The idea behind a connection-graph strategy is to first, prior to trying to prove anything by resolution, lay out the possible "resolvings out" as a graph. Thus suppose our clauses are:

$$p1 \vee p2 \vee p3$$
$$\neg p3 \vee p4$$
$$\neg p4 \vee \neg p1$$

We draw connections between the literals which might resolve out, thus:



Any clause which contains a literal that is unconnected cannot possibly lead to the derivation of the null clause, and so is deleted, along with any of its connections. So, in the above example, we delete the first clause and its connections, leaving:




But these too now have unconnected literals and thus are to be deleted. If there were no other clauses, we would know prior to starting the resolution portion of an attempted proof that it is not a theorem and hence we would not start. Using this technique, we get a theorem prover which, in a large number of cases, performs better than the usual resolution provers. The most advanced resolution prover we know of uses this connection-graph technique: it is that of J. Siekmann and his associates at the Universität Karlsruhe. However, even it is unable to show the following (predicate logic) argument as valid. (These remarks and the following argument are due to Len K. Schubert who presented this argument to Siekmann in 1978. We do not know the present status of the Karlsruhe theorem prover.)

> Every animal either likes to eat all plants or all animals much smaller than itself that like to eat some plants. Wolves, foxes, birds, caterpillars, and snails are animals, and there are some of each of them. Also there are some grains, and grains are plants. Caterpillars and snails are much smaller than birds, which are much smaller than foxes, which are in turn much smaller than wolves. Wolves do not like to eat foxes or grains, while birds like to eat caterpillars but not snails. Caterpillars and snails like to eat some plants. Therefore there is an animal that likes to eat a grain-eating animal.

(In passing here, we might point out that resolution provers require input in clause form — skolem normal form and the result in conjunctive normal form. As the above example demonstrates, it is not always a trivial task to transform ordinary language argumentation into clause form; yet when resolution theorists talk about the efficiency of their programs, they rarely mention the added effort required to massage the natural data into suitable input. Our theorem prover will accept any well-formed formula of first-order predicate logic.)

Bledsoe (1971) took what we think was a giant step forward in mechanical theorem proving when he introduced "splitting and reduction" techniques. These techniques are the basis for a human-like natural deduction system: they give overall strategies for proving theorems, which strategies depend upon the "main connective" of the formula to be proved. Thus for example, the strategy for proving a conjunction is to prove each of its conjuncts separately (rather than a resolution procedure's attempt to prove that the disjunction of the negations of each conjunct leads to the null resolvent). The strategy for proving a biconditional is to prove each conditional separately; the strategy for proving a conditional is to assume the antecedent true and attempt to prove the consequent — and so on. In 1971, Bledsoe used some of these techniques to "simplify" the formula to be proved, but then the final step for each subproblem was to do a resolution proof. We think, as in Bledsoe *et al* (1972), that this takes away a lot of the "human qualities" from the theorem prover. In this latter article, the resolution subsection was replaced by a procedure called IMPLY which used "forward chaining" and "backward chaining," in addition to some of the more usual resolution techniques. We have already mentioned above that we think forward and backward chaining by themselves are not very likely candidates for theorem proving because they require an immense amount of storage. We find furthermore that the mixture of resolution procedures

still with it takes us away from human-oriented theorem proving. Finally, it appears that the 1972 system cannot prove that:

$$((p \,\&\, (q \rightarrow r)) \rightarrow s)$$

is equivalent to:

$$((\neg p \vee (q \vee s)) \,\&\, (\neg p \vee (\neg r \vee s)))$$

which ours does, easily.

We think that the flaw with the Bledsoe systems is that the heuristic techniques were not followed far enough. We have found, with the complete use of the heuristics described below, that we could prove all theorems of the propositional logic and first-order predicate logic (without function symbols other than constants and without identify — two areas we have not yet tried to implement) found in Kalish and Montague (1964) and Thomason (1972). In addition, we can prove some theorems which are not in them, but which have cropped up in the mechanical theorem proving literature — such as $(p \vee \neg \neg \neg p)$ from the Logic Theorist, the Schubert argument from connection-graph resolution, and the example just given from Bledsoe.

## THINKER: A HEURISTIC-BASED THEOREM PROVER

We wish now to describe THINKER. We can divide the description into two components: the heuristics employed and the implementation of them. We shall skip lightly over the latter, for we think it of little interest to know that, *e.g.*, a doubly-linked circular list of all antecedent formulae with a '$\rightarrow$' as main connective was kept. But there are some features of the implementation which deserve to be mentioned, since they show how we have solved some of the problems of earlier theorem provers. First among these is that we used a version of SNOBOL (namely SPITBOL) rather than the more usual LISP-based languages. SNOBOL's basic operation is pattern matching, where these patterns can be as complex as one can describe (*e.g.*, recursive patterns). We are thus able to directly compare, for example, whether formula A is the negation of formula B by simply asking:

A ('$\neg$' B)

The blank after A is "match the pattern following" (which is what is in the parentheses) and the blank inside the parentheses indicates that one is to pre-concatenate a '$\neg$' to the pattern B. This is both *a.* in keeping with our syntactic view of logic as manipulation of symbols rather than with semantic structures, and *b.* in accordance with our intuitive psychological position that people attend to these patterns when constructing logical proofs.

As a side benefit, we get to store our formulae as strings, and this is less consuming than storing tree-like structures, as other provers usually do. (We could expand further on this, especially point *b.*, indicating how our patterns are essentially akin to human thought: "Ignore everything except to notice that the formula is a conditional that is universally quantified. Ignore the particular variable of quantification and only attend to the rest of the structure of the antecedent to see whether any

other line has this same structure when its variable is similarly ignored. *Etc.*" But we will not dwell on this here.)

Another feature of SNOBOL is that it has a primitive data structure called a TABLE, which is like an ARRAY, except we can access any element by using a string rather than an integer as an index. Thus, if we wish to know whether a certain formula is an antecedent line, we merely use that formula as an index and check directly. Again, this is much more efficient than the usual sequential search, even when the latter is augmented by graph-theoretic features; and it allows us to cut down our storage requirements.

One final feature should be mentioned here, and that is our use of TEMPLATES. When a line is added to the proof (when it becomes an antecedent line) we store information as to what type of line it is. If, for example, it is the conditional (A→B), we store these two pieces of information: *1.* it is a conditional with antecedent A; *2.* it is a conditional with consequent B. These are stored as the strings (A→@) and (@→B), where @ is a kind of metavariable indicating "some subformula or other." Since these are strings, we can use all the apparatus mentioned above about TABLES to access these "metaformulae." As an example, if we are trying to prove B, we might look to this TEMPLATE table, directly accessing by means of the string '(@→B)', to see if there is such a formula type in the antecedent lines. The information in this table is what formula @ is; and we therefore know exactly what this conditional in the Antecedent Lines TABLE (this table is called ANTELINES) is; so we can directly see whether the formula @ is also in ANTELINES. If it is, we perform a MP and add B to the ANTELINES, which constitutes a proof of the formula to be shown. It is easy to see that all this direct accessing solves the problems which have plagued the earlier theorem provers from the time of the Logic Theorist. Indeed, we think this breakthrough is one of the most important innovations of THINKER.

Having said that much about the implementation of THINKER, let us now turn to the heuristics involved. In fact, the heuristics are extremely simple and can be quickly described. What is surprising is that these simple heuristics, together with the implemention described above, will prove all that it does.

A Kalish and Montague proof has two structurally distinct parts to it. First is a stack of goals (formulae to be proved: the "show" lines detailed above) and second is a sequence of antecedent lines. We are always working at proving the most recently added goal, by adding more and more ANTELINES. When that goal is proved, it becomes an ANTELINE (= "cancelled") and all lines which had been added to ANTELINE after the goal was added to the goal stack get deleted (= "boxed"). When the first-to-be-added goal becomes antecedent, the proof is finished. So, when a theorem to be proved is entered, it becomes the first goal; since there are no ANTELINES to work with, it cannot be proved yet. THINKER then, on the basis of the theorem's main connective, decides what to do. Its choices are *a.* make an assumption (which becomes an ANTELINE), or *b.* add other goals. If the main connective is '&', '↔', or a universal quantifier, it will add a more simple goal and recursively call the whole set of heuristics on this new goal. When this simpler goal is proved, it uses it to prove the original goal. (These are Bledsoe's "splitting heuristics.") Otherwise it makes an assumption: if the main connective is '→', it assumes the antecedent; otherwise it assumes the negation. (Some of these are embodiments of Bledsoe's "reduction heuristics.") This is how proofs get started. From this point on, it uses the following heuristics.

*1.* If there are no antecedent lines (see *b* of last paragraph), it reapplies the "splitting and reduction" procedure.

*2.* ONESTEP(X) checks whether there is a rule of inference which uses X (some particular antecedent line), one application of which will prove the most recent

goal. This is rather simple to implement, since there are but a small number of rules of inference, and their working depends only on the structure of X and the most recent goal. Every time we add an ANTELINE to the proof, we call ONESTEP(X); if it succeeds, we "cancel" the most recent goal and "box."

3. SIMPLEPROOF(X) checks whether there are any two ANTELINES which, if one rule of inference were applied to them, would allow us to be able to add X (a particular formula we want to have, but not necessarily the most recent goal) as an ANTELINE. This is simple to do, since the "one application of a rule" applied to ANTELINES to yield X makes a rather limited search.
4. TRYRULES is a "blind" procedure which attempts to apply the propositional rules of inference and existential instantiation to ANTELINES. Each time a line X is added by this, ONESTEP(X) is called, which is one way to terminate TRYRULES. Another way to terminate it is if an ANTELINE is added to which heuristic *5* is applicable. Of course, TRYRULES and SIMPLEPROOF add lines to which TRYRULES is again applicable.
5. TRYNEGFLA is a rather clever strategy which — when more direct approaches fail — will search ANTELINES for an occurrence of the negation of a conditional and add the unnegated conditional to the goals. (It also looks for negations of disjunctions and adds one of the disjuncts to the goals; for negations of biconditionals, and negations of conjunctions. It then adds appropriate goals.) If this is successfully proved, the resulting contradiction will allow us to "cancel" the most recent goal.
6. TRYCHAINING: When other strategies fail, we look for a conditional in ANTELINES for which we do not have the consequent. We add, as a goal, the antecedent of the conditional. (If successful, we can perform a MP we couldn't before, so we can again TRYRULES). If this fails, and the negation of the antecedent isn't in ANTELINES, we try to add the negation of the consequent as a goal (to do a MT). A similar strategy applies to disjunctions and MTP.
7. HELP. When THINKER fails, either because the formula to be proved is not a theorem or because its heuristics are inadequate to prove it, it displays the proof as thus far constructed and requests the user to enter another line (either an ANTELINE or a new goal). It adds this to the proof with an appropriate annotation, and once again attempts the proof.
8. PROOF is the overall monitor of these other heuristics. It adds goals, calls the lower level heuristics as necessary, and recursively calls itself as new goals are added.

## AN EXAMPLE

We give here a moderately simple, propositional calculus example which illustrates how THINKER works.

The example is to show the equivalence between disjunction and the conditional: $((P \vee Q) \leftrightarrow (\neg P \rightarrow Q))$. This is put on the goal stack. Since it is a biconditional, PROOF adds a conditional to the goals: $((P \vee Q) \rightarrow (\neg P \rightarrow Q))$, and calls itself recursively. Since this is a conditional, it assumes '(PvQ)' and asks whether ONESTEP('(PvQ)') will prove the most recent goal. The answer is no, so it asks whether SIMPLEPROOF('$(\neg P \rightarrow Q)$'). Again no, so it adds '$(\neg P \rightarrow Q)$' as a goal, recursively calls itself, and assumes '$\neg P$'. It now calls ONESTEP('$\neg P$') to see whether '$(\neg P \rightarrow Q)$' is derivable from the ANTELINES. The answer is no, so it calls

SIMPLEPROOF('Q'). This succeeds (from the ANTELINES '(PvQ)' and ' ¬ P' by MTP) and so it adds 'Q' to ANTELINES, and then notices it can cancel the (¬ P→Q) goal. So it does (and deletes the ' ¬ P' ANTELINE), thus ending that recursive PROOF call. But this makes ((PvQ)→(¬ P→Q)) be proved, so it cancels that goal (and deletes the (PvQ) ANTELINE). PROOF then decides that to prove the original goal, it needs to prove ((¬ P→Q)→(PvQ)), so this is added to the goals. PROOF is recursively called and assumes (¬ P→Q). ONESTEP('(¬ P→Q)') fails, as does SIMPLEPROOF('(PvQ)'). So '(PvQ)' is added to the goals, and PROOF called recursively. It assumes '¬(PvQ)'. At this stage the proof looks like:

```
1.   Show ((P ∨ Q) ↔ (¬P → Q))
2.     *Show ((P ∨ Q) → (¬P → Q))
3.       (P ∨ Q)                    Assume
4.       *Show (¬P → Q)
5.         ¬P                       Assume
6.         Q                        3, 5MTP
7.     Show ((¬P → Q) → (P ∨ Q))
8.       (¬P → Q)                   Assume
9.       Show (P ∨ Q)
10.        ¬(P ∨ Q)                 Assume
```

No rules of inference apply to our three ANTELINES (#2, 8, 10). PROOF notices that line 10 is the negation of a disjunction and so calls TRYNEGFLA, which adds 'P' as a goal and calls PROOF recursively, ' ¬P' is assumed, and a MP is performed with the lines 8 and 12, yielding 'Q'. ONESTEP('Q') is called and succeeds, since by doing ADD on it, we generate a contradiction and hence can cancel our most recent goal. Having proved P and thus adding it to ANTELINES, PROOF notes that ONESTEP('P') allows us to prove the most recent goal (by ADD). But this most recent goal, '(PvQ)', was the consequent of the previous goal, and so that goal too is proved. We are now on the topmost recursion, trying to prove line 1, and we have two ANTELINES (#2, 7). We apply the rule CB to give us the final line #17, which allows us to cancel line 1, finish and print out the proof:

```
1.   *Show ((P ∨ Q) ↔ (¬P → Q))
2.     *Show ((P ∨ Q) → (¬P → Q))
3.       (P ∨ Q)                    Assume
4.       *Show (¬P → Q)
5.         ¬P                       Assume
6.         Q                        3, 5MTP
7.     *Show ((¬P → Q) → (P ∨ Q))
8.       (¬P → Q)                   Assume
9.       *Show (P ∨ Q)
10.        ¬(P ∨ Q)                 Assume
11.        *Show P
12.          ¬P                     Assume
13.          Q                      8, 12MP
14.          (P ∨ Q)                13, Add
15.          ¬(P ∨ Q)               10, R
16.        (P ∨ Q)                  11, Add
17.    ((P ∨ Q) ↔ (¬P ∨ Q))         2, 7CB
```

## CONCLUSION

While this example proof is rather simple, the heuristics employed are powerful. We think it surprising that with this meager set of heuristics, THINKER performs its task so well. Its success should encourage others to build natural-language-understanding systems and robotic-planning systems which incorporate similar heuristics. We hope that THINKER's success doesn't merely make critics say that heuristic theorem proving isn't "real" thinking, or that these aren't "real" heuristics. Anyone who tends in this direction might consider whether they think people do this any differently.

## REFERENCES

Bledsoe, W.W. "Splitting and Reduction Heuristics in Automatic Theorem Proving," *Artificial Intelligence,* 1971, *2,* 55–77.

Bledsoe, W.W. "Non Resolution Theorem Proving," *Artificial Intelligence,* 1977, *9,* 1–35.

Bledsoe, W.W., Boyer, R.S. and Henneman, W.H. "Computer Proofs of Limit Theorems," *Artificial Intelligence,* 1972, *3,* 27–60.

Chang, C.L. and Lee, R.C.T. *Symbolic Logic and Mechanical Theorem Proving.* New York: Academic Press, 1973.

Church, A. *An Introduction to Mathematical Logic.* Princeton: Princeton University Press, 1956.

Feigenbaum, J. and Feldman, J.A. *Computers and Thought.* New York: McGraw Hill, 1963.

Georgacarakos, G.M. and Smith, R. *Elementary Formal Logic.* New York: McGraw Hill, 1978.

Jeffrey, R. *Formal Logic: Its Scope and Limits.* Englewood Cliffs, NJ: Prentice-Hall, 1967.

Kalish, D. and Montague, R. *Logic: Techniques of Formal Reasoning.* New York: Harcourt, Brace, Jovanovich, 1964.

Kowalski, R. "A Proof Procedure Using Connection Graphs," *Journal of the Association for Computing Machinery,* 1974, *22,* 572–595.

Kowalski, R. *Logic for Problem Solving.* New York: Elsevier North Holland, 1979.

Newell, A., Shaw, J.C., and Simon, H.A. "Empirical Explorations with the Logic Theory Machine: A Case Study in Heuristics." 1957; reprinted as pp. 109–133 of Feigenbaum and Feldman 1963 (page references to reprint).

Newell, A. and Simon, J.C. *Human Problem Solving.* Englewood Cliffs, NJ: Prentice-Hall, 1972.

Thomason, R. *Symbolic Logic.* New York: Macmillan, 1972.