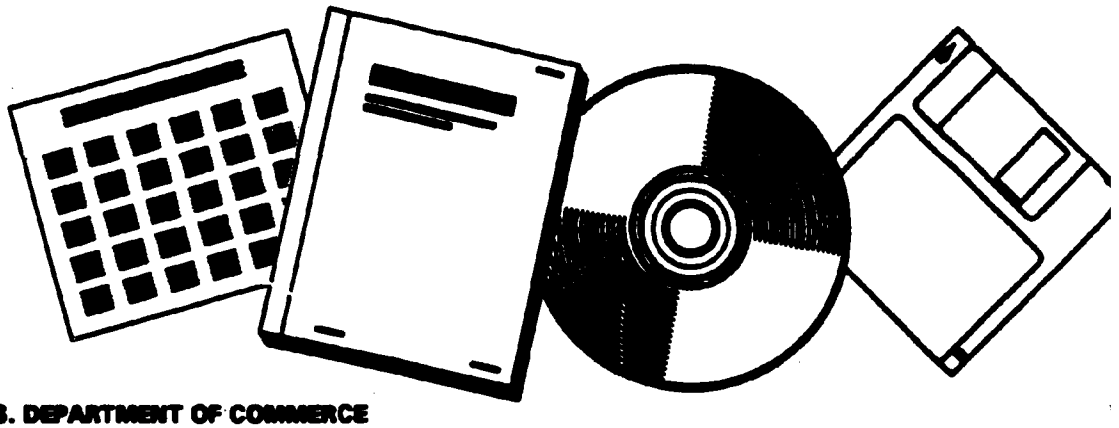
PB96-150883

NTIS
Information is our business.

AUTOMATIC DEDUCTION

STANFORD UNIV., CA

JUL 82



U.S. DEPARTMENT OF COMMERCE
National Technical Information Service

This document is complete as paginated by source. Refer to Index.

PB96-150883

October 1982
Also numbered: HPP-82-19

Report No. STAN-CS-82-937

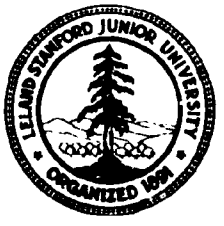
Automatic Deduction

by

Michael Ballantyne, W. W. Bledsoe, Jon Doyle
Robert C. Moore, Richard Pattis, Stanley J. Rosenschein

Department of Computer Science

Stanford University
Stanford, CA 94305



REPRODUCED BY:
U.S. Department of Commerce
National Technical Information Service
Springfield, Virginia 22161

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER STAN-CS-82-937; HPP-82-19 PB96-150883	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Automatic Deduction	5. TYPE OF REPORT & PERIOD COVERED technical, July 1982
	6. PERFORMING ORG. REPORT NUMBER STAN-CS-82-937; HPP-82-19
7. AUTHOR(s) Michael Ballantyne, W. W. Bledsoe, Jon Doyle, Robert C. Moore, Richard Pattis, Stanley J. Rosenschein (edited by Paul R. Cohen and Edward A. Feigenbaum)	8. CONTRACT OR GRANT NUMBER(s) MDA 903-80-C-0107
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science Stanford University Stanford, California 94305 U.S.A.	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency Information Processing Techniques Office 1400 Wilson Avenue, Arlington, VA 22209	12. REPORT DATE: July 1982 — 13. NO. OF PAGES: 64 15. SECURITY CLASS. (of this report): Unclassified
14. MONITORING AGENCY NAME & ADDRESS (if diff. from Controlling Office) Mr. Robin Simpson, Resident Representative Office of Naval Research, Durand 165 Stanford University	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this report) Reproduction in whole or in part is permitted for any purpose of the U.S. Government.	
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from report)	
18. SUPPLEMENTARY NOTES	
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report is reproduced from Chapter XII, "Automatic Deduction," of the Handbook of Artificial Intelligence (Vol. III, edited by Paul R. Cohen and Edward A. Feigenbaum). The chapter was written by Michael Ballantyne, W. W. Bledsoe, Jon Doyle, Robert C. Moore, Richard Pattis, and Stanley J. Rosenschein. Janice Aikins organized the chapter and edited most of the articles. This chapter on automatic deduction, also called automatic theorem proving, describes resolution and natural-deduction theorem proving, the Boyer-Moore theorem prover, nonmonotonic logic, and logic programming.

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19. KEY WORDS (Continued)

20 ABSTRACT (Continued)

Automatic Deduction

by

Michael Ballantyne, W. W. Bledsoe, Jon Doyle, Robert C. Moore,
Richard Pattis, and Stanley J. Rosenschein

Chapter XII of Volume III of the

Handbook of Artificial Intelligence

edited by

Paul R. Cohen and Edward A. Feigenbaum

This research was supported by both the Defense Advanced Research Projects Agency (ARPA Order No. 3423, Contract No. MDA 903-80-C-0107) and the SUMEX-AIM Computer Project under the National Institutes of Health (Grant No. NIH RR-00785). The views and conclusions of this document should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the National Institutes of Health, or the United States Government.

© 1982 by William Kaufmann, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. However, this work may be reproduced in whole or in part for the official use of the U.S. Government on the condition that copyright notice is included with such official reproduction. For further information, write to: Permissions, William Kaufmann, Inc., 95 First Street, Los Altos, CA 94022.

NTIS is authorized to reproduce and sell this report. Permission for further reproduction must be obtained from the copyright owner.

CHAPTER XII: AUTOMATIC DEDUCTION

FOREWORD

The Handbook of Artificial Intelligence was conceived in 1975 by Professor Edward A. Feigenbaum as a compendium of knowledge of AI and its applications. In the ensuing years, students and AI researchers at Stanford's Department of Computer Science, a major center for AI research, and at universities and laboratories across the nation have contributed to the project. The scope of the work is broad: About 200 short articles cover most of the important ideas, techniques, and systems developed during 25 years of research in AI.

Overview articles in each chapter describe the basic issues, alternative approaches, and unsolved problems that characterize areas of AI; they are the best critical discussions anywhere of activity in the field. These, as well as the more technical articles, are carefully edited to remove confusing and unessential jargon, key concepts are introduced with thorough explanations (usually in the overview articles), and the three volumes are completely indexed and cross-referenced to make it clear how the important ideas of AI relate to each other. Finally, the *Handbook* is organized hierarchically, so that readers can choose how deeply into the detail of each chapter they wish to penetrate.

This technical report is reproduced from Chapter XII, "Automatic Deduction," of the *Handbook* (Vol. III, edited by Paul R. Cohen and Edward A. Feigenbaum). The overview was written by Robert C. Moore, who also reviewed the other articles. W. W. Bledsoe provided the article on resolution theorem-proving and edited the article on natural deduction, which was prepared by Michael Ballantyne. Stanley J. Rosenschein wrote the article on logic programming; Richard Pattis, the article on the Boyer-Moore theorem prover; and Jon Doyle, the article on nonmonotonic logic. Janice Aikins organized the chapter and edited most of the articles.

A. OVERVIEW

A CENTRAL PROBLEM in AI research is how to make it possible for computers to draw conclusions automatically from bodies of facts. Any attempt to address this problem requires choosing an application, a representation for bodies of facts, and methods for deriving conclusions. This article provides an overview of the issues involved in drawing conclusions by means of deductive inference from bodies of commonsense knowledge represented by logical formulas. We first review briefly the history of automatic deduction—its origins, its fall into disfavor, and its recent revival. We show why deductive methods are necessary to solve problems that involve certain types of incomplete information and how supplying domain-specific control information offers a solution to the difficulties that previously led to disillusionment with automatic deduction. We discuss the relationship of automatic deduction to the new field of logic programming. Finally, we survey some of the issues that arise in extending automatic-deduction techniques to nonstandard logics.

Historical Background

Automatic deduction, or mechanical theorem-proving, has been a major concern of AI since its earliest days. At the first formal conference on AI, held at Dartmouth College in the summer of 1956, Newell and Simon (1956) discussed the Logic Theorist, a deduction system for propositional logic. Minsky was concurrently developing the ideas that were later embodied in Gelernter's theorem prover for elementary geometry (see McCorduck, 1979, p. 106; Gelernter, 1963). Shortly after this, Wang (1960) produced the first implementation of a reasonably efficient, complete algorithm for proving theorems in propositional logic.

Following these early efforts, the next important step in the development of automatic-deduction techniques was Robinson's (1965b) description of a relatively simple, logically complete method for proving theorems in first-order predicate calculus (see Article III.C1, in Vol. I). Robinson's procedure and those derived from it are usually referred to as *resolution* procedures (Article XII.B), because the basic rule of inference they use is the resolution principle:

From $(A \vee B)$ *and* $(\neg A \vee C)$, *infer* $(B \vee C)$.

Robinson's work had a major influence on two somewhat distinct lines of research. One of these was mathematical theorem-proving, which aims at providing practical tools for discovering new results in mathematics. (That line of research is not the main focus of this chapter, although Article XII.C is oriented in that direction.) But Robinson's work also had a major impact

on research into commonsense reasoning and problem solving. His ideas in this area brought about a rather dramatic shift in attitudes toward automatic deduction. The early attempts at automatic theorem-proving were generally thought of as exercises in expert problem solving: the Logic Theorist was regarded as an expert in propositional logic and Gelernter's program was considered an expert in geometry. However, the resolution method seemed powerful enough to make it possible to build a completely general problem-solver by describing problems in first-order logic and deducing solutions by a general proof procedure.

The idea of using formal logic as a representation scheme and deductive inference as a reasoning method was apparently first suggested as an approach to commonsense reasoning and problem solving by McCarthy in 1959, in his "Advice Taker" proposal (see McCarthy, 1968). Black (1968) made the first serious attempt to implement McCarthy's idea in 1964. Robinson's work provided encouragement for this approach, and a few years later Green (1969) carried out extensive experiments with a question-answering and problem-solving system based on resolution (see Article III.C1, in Vol. I, on the QA3 program).

The results of Green's experiments and several similar projects were disappointing, however. The difficulty was that, in the general case, the search space generated by the resolution method grows exponentially with the number of formulas used to describe a problem, so that problems of even moderate complexity cannot be solved in a reasonable time. Several domain-independent heuristics (e.g., *set of support;* see Article XII.B) were proposed to deal with this issue, but they proved too weak to produce satisfactory results.

It appears that these failures resulted principally from two constraints the researchers had imposed upon themselves: They attempted to use only uniform, domain-independent proof procedures, and they tried to force all reasoning and problem-solving behavior into the framework of logical deduction. Like a number of earlier ideas such as self-organizing systems and heuristic search, automatic theorem-proving turned out not to be the magic formula that would solve all AI problems at once. In the reaction that followed, however, not only was there a turning away from attempts to use deduction to create general problem-solvers, but there was also widespread condemnation of *any* use of logic or deduction in commonsense reasoning or problem solving. Arguments made by Minsky (1980, Appendix) and Hewitt (1975; Hewitt et al., 1973) seem to have been particularly influential in this regard.

Despite the disappointments of the late 1960s and early 1970s, there has recently been a revival of interest in deduction-based approaches to commonsense reasoning. This is apparent in the work of McDermott (1978), Doyle (1979, 1980), and Moore (1980a, 1980b); in the current work on nonmonotonic reasoning (Bobrow, 1980); and in recent textbooks by Nilsson (1980) and Kowalski (1979). To a large extent, this renewed interest seems to stem from

the recognition of an important class of problems that resist solution by any other method.

Why the Deduction Problem Will Not Go Away

If a description of a problem situation is *complete* in terms of the objects, properties, and relations relevant to the problem, we can answer any question by *evaluation*—deduction is unnecessary. To illustrate, suppose we have a knowledge base of personnel information for a company and we want to know whether there is any programmer who earns more than a vice-president earns. We could express this question in first-order logic as:

```
SOME (X,Y) ((TITLE(X) = PROGRAMMER) AND
            (TITLE(Y) = VICE-PRESIDENT) AND
            (SALARY(X) > SALARY(Y))) .
```

If we have recorded in our knowledge base the job title and salary of every employee, we can simply find the salary of each programmer and compare it with the salary of every vice-president. No deduction is involved in this process. On the other hand, we may not have specific salary information for each employee. Instead, we may have general information about classes of employees, such as:

All vice-presidents are managers.

```
ALL (X) ((TITLE(X) = VICE-PRESIDENT) →
         (CATEGORY(X) = MANAGER))
```

All programmers are professionals.

```
ALL (X) ((TITLE(X) = PROGRAMMER) →
         (CATEGORY(X) = PROFESSIONAL))
```

All professionals earn less than all managers.

```
ALL (X,Y) (((CATEGORY(X) = PROFESSIONAL) AND
            (CATEGORY(Y) = MANAGER)) →
            (SALARY(X) < SALARY(Y))) .
```

From this information we can *deduce* that no programmer earns more than any vice-president, although we have no information about the exact salary of any employee.

A representation formalism based on logic gives us the ability to express many kinds of generalizations, even when we do not have a complete description of the problem situation. Using deduction to manipulate expressions in the representation formalism allows us to make logically complex queries of a knowledge base containing such generalizations, even when we cannot evaluate a query directly. On the other hand, AI inference systems that are not based on automatic-deduction techniques either do not permit logically complex queries to be made or they answer such queries by methods that depend on the presence of complete information. For an AI system to handle

the kinds of incomplete information people can understand, it must at least be able to do the following:

1. Say that something has a certain property without saying which thing has that property:

$$\exists(X)\, P(X);$$

2. Say that everything in a certain class has a certain property without saying what everything in that class is:

$$\forall(X)\,(P(X) \rightarrow Q(X));$$

3. Say that at least one of two statements is true without saying which statement is true:

$$P \vee Q;$$

4. Say explicitly that a statement is false, as distinguished from simply not saying that it is true:

$$\neg(P).$$

Any representation formalism that has these capabilities will be, at the very least, an extension of classical first-order logic (see Article III.C1, in Vol. I), and any inference system that can deal adequately with these kinds of generalizations will have to have at least the capabilities of an automatic-deduction system. Thus, although AI rejected logic as a representation method and deduction as a reasoning method, AI systems that reason with incomplete information are actually equivalent to automatic-deduction systems.

The Need for Specific Control Information

As we remarked above, the fundamental difficulty with attempting to base a general, domain-independent problem-solver on automatic-deduction techniques is that there are too many possible inferences that can be drawn at any one time. Finding the inferences that are relevant to a particular problem can be an impossible task, unless domain-specific guidance is supplied to control the deductive process.

One kind of guidance that is often critical to efficient system performance is information about whether to use facts in a *forward-chaining* or *backward-chaining* manner. The deductive process can be thought of as a *bidirectional* search process (see Article II.C3d, in Vol. I), partly working forward from known facts to new ones, partly working backward from goals to subgoals, and meeting somewhere in between. Thus, if we have a fact of the form $(P \rightarrow Q)$, we can use it either to generate Q as a fact, given P as a fact, or to generate P as a goal, given Q as a goal. Early theorem-proving systems used every fact both ways, leading to highly redundant searches. More sophisticated methods

that eliminate these redundancies were gradually devised. Eliminating redundancies, however, creates choices as to which way facts are to be used. In the systems that attempted to apply only domain-independent control heuristics, a uniform strategy had to be imposed. Often the strategy was to use all facts in a backward-chaining manner only, on the grounds that this would at least guarantee that all the inferences drawn would be relevant to the problem at hand.

The difficulty with this approach is that the question of whether it is more efficient to use a fact for forward than for backward chaining depends on the specific content of that fact. For instance, according to the Talmud, the primary criterion for determining whether someone is Jewish is:

$$\forall(X)\,(Jewish(mother(X)) \rightarrow Jewish(X)).$$

That is, a person is Jewish if his or her mother is Jewish. Suppose we were to try to use this rule for backward chaining, as most uniform proof procedures would. It would apply to any goal of the form `JEWISH(X)`, producing the subgoal `JEWISH(MOTHER(X))`. This expression, however, is also of the form `JEWISH(X)`, so the process would be repeated, resulting in an infinite descending chain of subgoals:

```
GOAL: JEWISH(MORRIS)
GOAL: JEWISH(MOTHER(MORRIS))
GOAL: JEWISH(MOTHER(MOTHER(MORRIS)))
GOAL: JEWISH(MOTHER(MOTHER(MOTHER(MORRIS))))
  ⋮
```

If, on the other hand, we use the rule for forward chaining, the number of applications is limited by the complexity of the fact that originally triggers the inference:

```
FACT: JEWISH(MOTHER(MOTHER(MORRIS)))
FACT: JEWISH(MOTHER(MORRIS))
FACT: JEWISH(MORRIS) .
```

It turns out, then, that the efficient use of a particular fact often depends on exactly what that fact is and also on the context of other facts in which it is embedded. Many examples illustrating this point are given by Kowalski (1979) and Moore (1980a), involving not only the distinction between forward and backward chaining but other control decisions as well.

Since specific control information needs to be associated with particular facts, the question arises as to how to provide it. The simplest way is to embed it in the facts themselves. For instance, the distinction between forward and backward chaining can be encoded by having two versions of implication, for example, $(P \rightarrow Q)$ to indicate forward chaining and $(Q \leftarrow P)$ to indicate backward chaining. This approach originated in the distinction made in

the programming language PLANNER (see Article VI.A. in Vol. II) between antecedent and consequent theorems. A more sophisticated approach is to make certain decisions (such as whether to use a fact in the forward or backward direction) themselves questions for the deduction system to reason about, by using "meta-level" knowledge. The first detailed proposal along these lines appears to have been made by Hayes (1973), while experimental systems have been built by McDermott (1978) and de Kleer et al. (1979), among others. Weyhrauch (1980) has perhaps done the most to explore the kind of system architecture in which this sort of reasoning would be possible.

Theory Formation and Logic Programming

Another factor that can greatly affect the efficiency of deductive reasoning is the way in which a body of knowledge is formalized. That is, logically equivalent formalizations can have radically different behavior when used with standard deduction techniques. For example, we could define the relation ABOVE as the transitive closure of ON in at least three ways:

$$\forall (X,Y)\ (\text{ABOVE}(X,Y) \leftrightarrow (\text{ON}(X,Y)\ \text{OR}\ \exists (Z)\ (\text{ABOVE}(X,Z)\ \text{AND}\ \text{ON}(Z,Y)))) .$$

$$\forall (X,Y)\ (\text{ABOVE}(X,Y) \leftrightarrow (\text{ON}(X,Y)\ \text{OR}\ \exists (Z)\ (\text{ON}(X,Z)\ \text{AND}\ \text{ABOVE}(Z,Y)))) .$$

$$\forall (X,Y)\ (\text{ABOVE}(X,Y) \leftrightarrow (\text{ON}(X,Y)\ \text{OR}\ \exists (Z)\ (\text{ABOVE}(X,Z)\ \text{AND}\ \text{ABOVE}(Z,Y)))) .$$

(These formalizations are not quite equivalent, as they allow for different possible interpretations of ABOVE if infinitely many objects are involved. They are equivalent, however, if only finitely many objects are being considered.)

Each of these formalizations will produce different behavior in a standard deduction system, no matter how we make local control decisions of the kind discussed in the previous section. Kowalski (1974) noted that choosing among such alternatives involves decisions similar to those made when writing programs in a conventional programming language. In fact, he observed that there are ways to formalize many functions and relations so that applying standard deduction methods will have the effect of executing them as computer programs. These observations have led to the development of the field of *logic programming* (Kowalski, 1979) and the creation of new computer languages such as PROLOG (Warren, Pereira, and Pereira, 1977). Such developments are discussed in Article XII.F.

Automatic Deduction in Nonstandard Logics

So far, we have discussed automatic deduction for classical first-order logic only. Many commonsense concepts, however, are most naturally treated

in either higher order or nonclassical logics. This presents a problem, because classical first-order logic is the most general logic for which the techniques of automatic deduction are at all well developed. It turns out, though, that there are a number of techniques for reformulating representations in nonstandard logics in terms of logically equivalent representations in classical first-order logic.

Higher order logic differs from first-order logic in that it allows quantification over properties and relations as well as individuals. That is, if we have a first-order logic that allows us to make statements about all physical objects, the corresponding second-order logic would allow us to make statements about all properties of and relations among physical objects; a third-order logic would allow us to make statements about properties of and relations among these properties and relations; and so forth.

In some cases, the transition from first-order to higher order logic presents fewer difficulties than might at first appear. In fact, the standard deductive procedures for first-order logic also work for higher order logic, except that general predicate abstraction is not performed; that is, these procedures will not construct predicates out of arbitrary complex formulas. If *John is a man* is represented as `MAN(JOHN)`, the predicate `MAN` can be retrieved when we ask the second-order question, *What properties does John have?* All the deduction system has to do is match `X(JOHN)` against `MAN(JOHN)` and return `MAN` as the value of the variable `X`. But from the assertion that John is either a butcher or a baker, represented as

```
BUTCHER(JOHN) OR BAKER(JOHN) ,
```

the system could not infer, without using predicate abstraction, that John has the disjunctive property of being a butcher-or-baker. The system would have to recognize that this complex expression could be reformulated as a one-place predicate applied to `JOHN`,

```
(LAMBDA (Y) (BUTCHER(Y) OR BAKER(Y)))(JOHN) ,
```

which is of the right form to match `X(JOHN)`.

If this sort of predicate abstraction is not required, standard first-order deduction techniques are sufficient. There has been some work extending the standard techniques to handle the more general case (e.g., Huet, 1975), but this makes the deduction problem much harder because of the combinatorics of all the different ways predicate abstraction may be performed.

Another problem commonly encountered is how to do automatic deduction in logics that allow *intensional* operators. These are operators, such as BELIEVE and KNOW, that produce sentences whose truth values depend fully on the meanings, not just the truth values, of their arguments. Classical logic is purely *extensional*, because the truth value of a complex formula depends only on the extensions (denotations, referents) of its subexpressions. The extension of a formula is considered to be its truth value, so the operator OR

is extensional because the truth of (*P or Q*) depends only on the truth of *P* and the truth of *Q*; no other properties of *P* and *Q* matter. The operator BELIEVE, on the other hand, is intensional because the truth of *A believes that P* depends generally on the meaning of *P*, not just on its truth value.

Many of the rules of classical logic, such as substitution of equals for equals, do not apply within the scope of an intensional operator. To use a classic example, since *the morning star* and *the evening star* refer to the same object, it must be the case that *The morning star is Venus* is true if and only if *The evening star is Venus* is true. However, it might be that *John believes the morning star is Venus* is true, but that *John believes the evening star is Venus* is false because, although the two embedded sentences have the same truth value, they differ in meaning.

Fortunately, many of the difficulties presented by intensional operators can be overcome by reformulating the statements in which they occur. There are a number of methods for doing this, but one that is particularly elegant is to reformulate intensional operators in terms of their *possible-world semantics* (Kripke, 1971; Hintikka, 1971). The idea is that, rather than talking about what statements a person believes, we talk instead about what states of affairs, or possible worlds, are compatible with what he believes. Essentially, *A believes that P* is paraphrased as *P is true in every world that is compatible with what A believes*. This can be expressed in ordinary first-order logic by making all predicates and functions depend explicitly on the particular possible world they are evaluated in. The failure of equality substitution in the preceding example is then accounted for by noting that what John believes depends on what is true in *all* possible worlds that are compatible with what he believes, but an assertion that the morning star and the evening star are the same is a statement only about the *actual* world. Application of this idea to reasoning about intensional operators in AI systems has been explored in depth by Moore (1980b).

Finally, a type of nonstandard logic that has received much recent attention is *nonmonotonic logic*. Minsky (1980, Appendix) has noted that the treatment of commonsense reasoning as purely deductive ignores one of its crucial aspects—the ability to retract a conclusion in the face of further evidence. A frequently cited example is that, if we know something is a bird, we normally assume it can fly. If we find out that it is an ostrich, however, we will withdraw that conclusion. This sort of reasoning is called nonmonotonic because the set of inferable conclusions does not increase monotonically with the set of premises as in conventional deductive logics. The addition of the premise that something is an ostrich results in removing the conclusion that it can fly. While many procedures have been implemented that support this type of reasoning, their theoretical foundations are questionable. Most of the recent work on nonmonotonic logic (Bobrow, 1980; see Article XII.E) has thus been directed at developing a coherent logical basis for this kind of reasoning.

References

McCarthy (1968), Black (1968), and Green (1969) discuss formal logic as a representation scheme and deductive inference as a reasoning method for commonsense reasoning and problem solving. This theme is amplified in two readable texts by Nilsson (1971, 1980). For references on some of the other topics discussed in the overview, see the reference sections of the subsequent articles.

B. THE RESOLUTION RULE OF INFERENCE

ONE of the best known methods of automatic theorem-proving is the *resolution* procedure introduced by J. A. Robinson (1965b). In this article, we describe the method, present some examples, and discuss extensions to it.

Derivation of the Resolution Rule

The resolution method shows whether a theorem logically follows from its axioms. If a theorem does follow from its axioms, then the axioms and the *negation* of the theorem cannot all be true—the axioms and the negated theorem must lead to a contradiction. The resolution method is a form of proof by contradiction that involves producing new clauses, called *resolvents*, from the union of the axioms and the negated theorem. These resolvents are then added to the set of clauses from which they were derived, and new resolvents are derived. This process continues, recursively, until it produces a contradiction. Resolution is guaranteed to produce a contradiction if the theorem follows from the axioms. The simple *resolution rule* that produces resolvents is derived in the following paragraphs.

By the expression $(P \rightarrow Q)$ we mean *If P is true, then Q is true;* for example, *John is a boy* $\rightarrow$ *John is male.* A central rule of inference in logic is *modus ponens:*

$$(((P \rightarrow Q) \text{ and } P) \vdash Q),$$

which means that if $(P \rightarrow Q)$ is true and if P is true, then we can conclude that Q is true. An extension of this is the *chain rule:*

$$((P \rightarrow Q) \text{ and } (Q \rightarrow R) \vdash (P \rightarrow R)).$$

When the implications in the chain rule are rewritten in their logically equivalent form $(\neg P \vee Q)$, the chain rule becomes

$$(\neg P \vee Q) \text{ and } (\neg Q \vee R) \vdash (\neg P \vee R),$$

which can be written as:

$$\frac{(\neg P \vee Q)}{(\neg Q \vee R)}$$
$$\overline{(\neg P \vee R)}.$$

There is an apparent cancellation of the Q and $\neg Q$. The disjunctions $(\neg P \vee Q)$, $(\neg Q \vee R)$, and $(\neg P \vee R)$ are called *clauses*, and $(\neg P \vee R)$ is called the *resolvent* of $(\neg P \vee Q)$ and $(\neg Q \vee R)$.

Implications in this simple form, called *clause form*, can be resolved against each other; two clauses can be resolved to a single one. The heart of the resolution proof method is to negate the theorem to be proved and then to simplify and resolve clauses until a contradiction is found.

An Example

As an example of resolution, consider proving that $(D \vee E)$ follows from $(A \rightarrow C \vee D) \wedge (A \vee D \vee E) \wedge (A \rightarrow \neg C)$. The first step is to negate the theorem: $(\neg(D \vee E))$. This is logically equivalent to $(\neg D \wedge \neg E)$. The next step is to convert the axioms and theorem to *clauses*. The procedures for this are explained in the last section of this article and in several texts (e.g., Nilsson, 1980); all we need to know here is that the implication $(A \rightarrow B)$ can be rewritten as the equivalent clause $(\neg A \vee B)$.

The axioms are:

$$(A \rightarrow C \vee D) \wedge$$
$$(A \vee D \vee E) \wedge$$
$$(A \rightarrow \neg C).$$

They are rewritten as the clauses, and the theorem is added to the list:

$$(\neg A \vee C \vee D) \wedge$$
$$(A \vee D \vee E) \wedge$$
$$(\neg A \vee \neg C)$$
$$(\neg D \wedge \neg E).$$

The $\wedge$ conjunctions are dropped, leaving five clauses:

1. $(\neg A \vee C \vee D)$
2. $(A \vee D \vee E)$
3. $(\neg A \vee \neg C)$
4. $(\neg D)$
5. $(\neg E)$.

If the theorem follows from its axioms, the axioms and the negation of the theorem cannot all be true. Consequently, a contradiction must be implicit in the five clauses just derived; they cannot all be true simultaneously. The purpose of resolution is to find the contradiction. We will resolve clauses against each other until a contradiction "drops out":

1. $(\neg A \vee C \vee D)$
2. $(A \vee D \vee E)$

Resolution 1: $(C \vee D \vee E)$ — *$\neg A$ and A cancel each other.*

2. $(A \vee D \vee E)$
3. $(\neg A \vee \neg C)$

Resolution 2: $(D \vee E \vee \neg C)$ — *$\neg A$ and A cancel each other.*

Resolution 1. $(C \vee D \vee E)$
Resolution 2. $(D \vee E \vee \neg C)$

Resolution 3: $(D \vee E)$ — *$\neg C$ and C cancel each other.*

Resolution 3. $(D \vee E)$
4. $(\neg D)$

Resolution 4: (E) — *$\neg D$ and D cancel each other.*

Resolution 4. (E)
5. $(\neg E)$

CONTRADICTION

This illustrates the process by which we determine that clauses and their resolvents cannot all be true simultaneously.

The example just presented is from *propositional logic*. Now let us consider first-order *predicate calculus*, where *variables*, *predicates*, *quantifiers*, and *functions* are permitted (see Article III.C1. in Vol. I, for a discussion of logics). The expression $P(x)$ means *P is true for x*. For example, $P(x)$ might mean *x is a positive number*, so that $P(2)$ is true, whereas $P(-3)$ is false. Or $P(x)$ might mean that *x is a boy*, in which case we would expect $P(John)$ to be true and $P(Peggy)$ to be false.

We will use the notation $\forall x\ P(x)$ and $\exists x\ P(x)$ to mean *For all x $P(x)$* and *For some x $P(x)$*, respectively. The first form is called a *universal quantification*, since it conveys the meaning that the clause is true for all objects; the second is called an *existential quantification*, since it says that the clause is true for at least one object. For example,

$$\forall x\,(N(x) \rightarrow x^2 \geq 0), \text{ and}$$
$$\exists x\,(N(x) \wedge x < 0)$$

are true formulas. The first says that if x is a number, then the square of all x is either positive or zero, whereas the second says that there is at least one

object that is a number and is negative. Notice that $\neg\forall x\ P(x)$ is equivalent to $\exists x\ \neg P(x)$, and $\neg\exists x\ P(x)$ is equivalent to $\forall x\ \neg P(x)$.

It is also possible to have function symbols such as f and g. For example, $f(x)$ can mean *father of* x. Thus, if $M(x)$ means x *is a male*, then $M(f(x))$ is always true.

Two complications arise when proving theorems with variables, quantifiers, predicates, and functions. One is getting them into clause form; the other is the process of *unification*. Converting predicate logic to clause form is formally straightforward (see the last section of this article). However, it is important to understand the conceptual operations as well as the formal ones, especially those associated with eliminating quantifiers. To eliminate existential quantifiers, we simply choose a constant; for example, $\exists x\ P(x)$ is replaced by $P(a)$. We instantiate the claim that an x exists by choosing a particular a to take its place. However, if an existential quantifier is within the scope of a universal quantifier, there is the possibility that the x that exists somehow depends on the identity of the universally quantified variable. Thus, we cannot replace it with an arbitrary constant. To account for this, whenever an existential quantifier occurs within the scope of a universal quantifier, its variable is replaced with a function of the universally quantified variable. For example, $\forall x \exists y\ P(x, y)$ is rewritten as $\forall x\ P(x, f(x))$, denoting that the second argument of the predicate P is a function of the first. In this example, f is called a *skolem function*, and $f(x)$ is called a *skolem expression*.

We have discussed the rationales for eliminating existential quantifiers. Universal quantifiers are simply dropped from clause form, because after existentially quantified variables have been replaced by constants or skolem functions, we may assume that the remaining variables are universally quantified. In the previous example, y was replaced by a skolem function and x is assumed to be universally quantified; thus, the quantifier $\forall$ is deleted, resulting in the clause $P(x, f(x))$.

The other complication in proving theorems in predicate calculus arises during resolution itself. Recall that during resolution we would have constants "canceling" each other out; for example, $\neg A \vee B$ and $A \vee C$ would resolve to $B \vee C$ after canceling A and $\neg A$. But how are resolvents to be produced when there are variables and skolem functions? For example, does $P(a)$ cancel $\neg P(x)$ in the following resolution?

$$\frac{\begin{array}{c}\neg P(x) \vee Q(x) \text{ and} \\ P(a) \vee R(x)\end{array}}{Q(a) \vee R(x)}$$

In this case, the answer is yes: $P(a)$ cancels $\neg P(x)$, because the expression $\neg P(x)$ is claiming that there is no x for which $P(x)$ is true (recall that x is universally quantified), and $P(a)$ is claiming that there *is* an object a for which $P(a)$ is true. This is an example of *unification*, the process of deciding whether

the arguments of predicates are comparable for the purpose of resolution, and, if they are comparable, what common *substitution instance* should be used. In this case, the substitution instance was a; it replaces all instances of x, including that in the predicate Q. The process of unification is analogous to that of finding a common denominator for fractions: In order to make comparisons between numbers expressed as $x/3$ and numbers expressed as $x/17$, each is re-expressed as $x/51$. Similarly, there is a *unification algorithm* that finds a common substitution instance for the arguments of predicates.

With these preliminaries over, we can now proceed to examples of resolution theorem proving in the predicate calculus.

The first step is, again, to negate the theorem and then put the axioms and the theorem in clause form:

$(\neg P(a) \wedge \forall x\,(P(x) \vee Q(f(x))))$	(Axiom)
$\exists z\,Q(z)$	(Theorem)
$\forall z\,\neg Q(z)$	(Negated Theorem)

In this case, a is a constant symbol, and there are no existential quantifiers and so no need for skolemization. Universal quantifiers are simply dropped. The $\wedge$ connectives are also dropped to yield three clauses:

1. $\neg P(a)$
2. $P(x) \vee Q(f(x))$
3. $\neg Q(z)$.

These are resolved against each other as follows:

1. Clause 1 and clause 2 are resolved to produce $Q(f(a))$: the substitution is a for x, or a/x.
2. $Q(f(a))$ is resolved against clause 3 to yield a contradiction: the substitution is $f(a)$ for z, or $f(a)/z$.

Since a contradiction is produced, we can conclude that the theorem followed from its axioms.

Another example is proving that there is always a number greater than another number from the axiom that a number is less than its successor. (In this case, *infix* arithmetic functions are used in the clauses: they could equally well be written in *prefix* notation; e.g., $\forall t < (t, \text{PLUS}(t, 1))$.)

$\forall t\,(t < t + 1)$	(Axiom)
$\forall x \exists y\,(x < y)$	(Theorem)

First we negate the theorem:

$$\exists x \forall y\,\neg(x < y).$$

Then, since x is an existentially quantified variable that is not within the scope of a universal quantifier, we replace it with a constant. This eliminates the

existential quantifier: universal quantifiers are simply dropped as before. The resulting clauses are:

1. $t < t + 1$
2. $\neg(a < y)$.

But this immediately results in a contradiction when a is substituted for t and $a + 1$ is substituted for y.

A final example illustrates skolemization:

$$\forall x \exists y\, P(x, y) \qquad \text{(Axiom)}$$
$$\exists z\, P(a, z) \qquad \text{(Theorem)}$$

where a is a constant. First, we negate the theorem, yielding $\forall z\, \neg P(a, z)$. Next, we eliminate quantifiers. Since $\exists y$ is within the scope of the universal quantifier $\forall x$, the variable y is replaced, not with a constant, but, instead, with a skolem function. Universal quantifiers are dropped as usual:

1. $P(x, g(x))$
2. $\neg P(a, z)$.

These clauses obviously resolve to a contradiction under the substitution $a/x,\ g(a)/z$.

It can be shown that resolution is *complete* for (i.e., can prove all theorems in) first-order predicate logic (Robinson, 1965b) and is *sound* (i.e., will not indicate that nontheorems are true).

Strategies

Although resolution is complete, it can be extremely time-consuming. As brought out in the overview (Article XII.A), resolution-based approaches to problem solving fell into disfavor for just this reason.

Several strategies have been proposed to minimize the branching factor of resolution proof trees. Several are discussed in detail in Nilsson (1980) and in Chang and Lee (1973), and, thus, only two are briefly discussed here.

Set-of-support strategy. When at least one parent of each resolvent is chosen from the negation of the theorem or from the set of clauses that are derived from it, a set-of-support strategy is being used. This strategy clearly restricts the number of clauses that can be resolved at any given time. It is usually more efficient than breadth-first search.

Linear-input-form strategy. This strategy involves choosing resolvents so that one resolvent is always from the base set (the set of original clauses). It is more efficient than the previous strategy, but it is not *complete*, which is to say that there are cases in which it will not find a contradiction when one exists. Nonetheless, the strategy is often used because of its simplicity and efficiency.

In addition to strategies designed to reduce the combinatorial explosion involved in resolution, other simplifications can be made. One is to eliminate *tautologies* from the set of clauses. A tautology is a trivially true clause containing the subexpression $A \vee \neg A$.

Converting a Formula to Clausal Form

A formula, F, to be proved by resolution must first be negated and converted to clausal form. It is assumed that F is a first-order formula that is fully quantified. Conversion to clausal form is done by a series of steps:

1. Negate F: Replace F by $\neg F$.
2. Remove $\rightarrow$ and $\leftrightarrow$ by replacing $(A \rightarrow B)$ by $(\neg A \vee B)$ and $(A \leftrightarrow B)$ by $((\neg A \vee B) \wedge (\neg B \vee A))$.
3. Move $\neg$ inward, using the rules:
$$\neg(\neg A) \equiv A,$$
$$\neg(A \wedge B) \equiv \neg A \vee \neg B,$$
$$\neg(A \vee B) \equiv \neg A \wedge \neg B,$$
$$\neg \forall x\, A(x) \equiv \exists x\, \neg A(x),$$
$$\neg \exists x\, A(x) \equiv \forall x\, \neg A(x).$$
4. Move $\forall$ and $\exists$ inward (optional).
5. Rename variables so that no two quantifiers quantify the same variables.
6. Exchange $\exists$ for skolem functions and then drop $\forall$'s (see below).
7. Convert to CNF (conjective normal form) by repeatedly applying De Morgan's Laws:
$$\neg(A \wedge B) \equiv \neg A \vee \neg B$$
$$\neg(A \vee B) \equiv \neg A \wedge \neg B.$$

In step 6, if $\exists y\, P(y)$ is within the scope of universal quantifiers $\forall x_1 \forall x_2 \ldots \forall x_n$, and not within the scope of any existential quantifier, then replace $\exists y\, P(y)$ by $P(f(x_1, \ldots, x_n))$, where f is a new function symbol (a skolem-function symbol). All universal quantifiers are then dropped from the formula. Thus,

$$\forall x\, \exists y \forall z\, \exists w\, P(x, y, z, w)$$

is replaced successively by

$$\forall x \forall z\, \exists w\, P(x, f_1(x), z, w)$$
$$\forall x \forall z\, P(x, f_1(x), z, f_2(x, z))$$
$$P(x, f_1(x), z, f_2(x, z)).$$

If $n = 0$, then y is replaced by a skolem constant y_0 (i.e., a function of 0 arguments).

It is usually faster to replace $\neg(P \rightarrow Q)$ by $(P \wedge \neg Q)$ *before* converting $(P \rightarrow Q)$ to $(\neg P \vee Q)$, when P is a large formula.

References

The resolution rule of inference was first described by Robinson (1965b). Resolution has been extended to handle the equality relation; this is discussed in Robinson and Wos (1969). This extension permits one to prove theorems such as $P(a) \wedge a = b \rightarrow P(b)$.

Strategies for speeding up resolution theorem proving have been discussed in several places. Wos, Robinson, and Carson (1965) discussed *set of support; hyper-resolution* was considered by Robinson (1965a); *locking* was the subject of Boyer's thesis (1971); and *SL-resolution* was discussed by Kowalski and Kuchner (1971). *Model elimination* was introduced by Loveland (1978). General texts on theorem proving are Loveland (1978) and Chang and Lee (1973).

Nilsson's two textbooks (1971, 1980) are clearly written introductions to, among other things, theorem proving as a problem-solving tool for AI systems.

C. NONRESOLUTION THEOREM PROVING

IN *nonresolution* or *natural-deduction* theorem-proving systems, a proof is derived in a goal-directed manner that is natural for the humans using the theorem prover. Natural-deduction systems represent proofs in a way that maintains a distinction between goals and antecedents, and they use inference rules that mimic the reasoning of human theorem-provers.

In resolution theorem-provers, no distinction is made between goals and antecedents. But in natural-deduction systems, the distinction is carefully maintained for the clarity that it brings to the proof process. For example, a natural-deduction system might display the following "worksheet" during a proof:

$$\begin{array}{l} H_1.\ P \\ H_2.\ (P \rightarrow Q) \\ H_3.\ (R \wedge Q \rightarrow S) \\ \hline C_1.\ Q \\ C_2.\ (R \rightarrow S). \end{array}$$

It indicates that H_1, H_2, and H_3 are three hypotheses and C_1 and C_2 are goals. A resolution system would represent the same situation uniformly with a set of clauses:

1. P
2. $\neg P \vee Q$
3. $\neg R \vee (\neg Q \vee S)$
4. $\neg Q \vee R$
5. $\neg Q \vee \neg S$.

Although these representations are logically equivalent, we have lost all information in the second one about *goals*—about what we want to prove.

The representation of proofs in natural-deduction systems is especially advantageous for man-machine interactive theorem-proving, in which a human is required to intervene occasionally to help with the proof. It also facilitates the implementation of semantic or domain-specific heuristics that help to guide the search.

However, the *clausal* representation has one powerful advantage: A proof can be derived with a single inference rule—the *resolution* rule. In contrast, natural-deduction systems have relatively complex inference rules that simulate the kinds of reasoning steps that humans use to develop proofs. For example, suppose we want to prove that Fred has a hot tub, and we know

that everyone who lives in California has a hot tub and that Fred lives in California:

Antecedents: $(\textit{Live-California}(\textit{Fred})) \wedge (\textit{Live-California}(X) \rightarrow \textit{Hottub}(X))$
Goal: $\rightarrow \textit{Hottub}(\textit{Fred})$.

To prove *Hottub(Fred)*, we scan the antecedents for anything that will enable us to conclude *Hottub(Fred)*, and, if we find such a hypothesis, we set up the subgoal of proving it. In this case, we can conclude *Hottub(Fred)* if we can prove (*Live-California*(X) $\rightarrow$ *Hottub*(X) and (*Live-California*(*Fred*)). So we set up the subgoal of proving *Live-California*(*Fred*). Formally, we can derive a *back-chain* rule of inference:

To prove $[H \wedge (A \rightarrow B) \rightarrow C]$:
If $(B \rightarrow C)$, then prove $(H \rightarrow A)$.

In the next section, we present several of the proof rules from the IMPLY system, developed at the University of Texas (Bledsoe and Tyson, 1975).

IMPLY

IMPLY views a conjecture to be proved as a conjunction of goals to be achieved, and it considers a goal achieved when it finds a *substitution* under which the goal is valid. A substitution is simply an assignment of terms to each variable in the conjecture. In other words, IMPLY considers a conjecture proved when it finds some object or objects for which the conjecture is valid. For example, the conjecture

$$(P(x) \rightarrow Q(x)) \wedge P(a) \rightarrow Q(a)$$

is valid for the substitution a/x; that is, if every x in the formula were replaced by a, then the statement would be a valid inference.

Let C be a conjecture we wish to prove and let H be the conjunction of hypotheses that, hopefully, imply C. IMPLY will attempt to find a substitution (θ) such that $(H \rightarrow C)(\theta)$ is a propositionally valid formula. For example, if H is

$$P(a) \wedge (P(x) \rightarrow Q(x))$$

and C is

$$Q(a),$$

then the substitution $(\theta) = a/x$ will make $(H \rightarrow C)(\theta)$ valid.

In the following discussion, we assume that all formulas are quantifier free. That is, before the proof process starts, all universal and existential quantifiers, $\forall$ and $\exists$, are removed by *skolemization* (see Article XII.B). Skolemization for both resolution and natural deduction is done in much the same way, except that the roles of $\forall$ and $\exists$ in natural deduction are the

opposite of their roles in resolution, because resolution is a refutation procedure and natural deduction is not. For example, for natural deduction, $[\forall x\, P(x) \rightarrow Q(a)]$ skolemizes to $[P(x) \rightarrow Q(a)]$ and $[H \rightarrow \exists x \forall y\, P(x, y)]$ skolemizes to $[H \rightarrow P(x, g(x))]$.

Formulas are submitted to IMPLY, which attempts to prove them by application of the rules discussed below. If F is a formula, $[F]$ denotes the value of IMPLY applied to F.

IMPLY rules. Some of the IMPLY proof rules are shown below.

1. MATCH: $[H \rightarrow C]$
   If $H(\theta) = C(\theta)$,
   then (θ)
   (the empty substitution is T).

This is the simplest of IMPLY's rules. The goal C is matched to the hypothesis H and, if a substitution can be found, that substitution is returned. For example, $(P(x) \rightarrow P(a))$ is MATCH because a substitution a/x makes H and C equal. The substitution is found by *unification* (see Article XII.B). MATCH would fail for the clause $(Q(x) \rightarrow P(a))$ because the predicates P and Q are different.

2. AND-SPLIT: $[H \rightarrow A \wedge B]$
   If $[H \rightarrow A]$ is (θ)
   and $[H \rightarrow B(\theta)]$ is (λ),
   then $(\theta)(\lambda)$.

If we want to prove that H implies A and B, we first prove that $(H \rightarrow A)$ for some substitution, and then, using that substitution in B, we prove that $(H \rightarrow B)$. For example, to prove $[P(x) \rightarrow P(a) \wedge (Q(x) \rightarrow P(a))]$, we obtain the substitution a/x when we prove $[P(x) \rightarrow P(a)]$, and that substitution is carried into the second step, namely, to prove $[P(x) \rightarrow (Q(a) \rightarrow P(a))]$. If, in proving this, we obtain another substitution, λ, then θ and λ are *composed* to produce a substitution under which the entire expression $[P(x) \rightarrow P(a) \wedge (Q(x) \rightarrow P(a))]$ is valid.

3. CASES: $[H_1 \vee H_2 \rightarrow C]$
   If $[H_1 \rightarrow C]$ is (θ)
   and $[H_2(\theta) \rightarrow C]$ is (λ),
   then $(\theta)(\lambda)$.

To prove that either of H_1 or H_2 implies C, we must prove that they both do. Thus, we attempt first to prove $[H_1 \rightarrow C]$ for some substitution, then $[H_2 \rightarrow C]$ under the previous substitution, and, if this second proof produces a substitution, the two are composed.

4. OR-FORK: $[A \wedge B \rightarrow C]$
   If $[A \rightarrow C]$ is (θ),
   then (θ);
   else $[B \rightarrow C]$.

To show that A and B imply C, we must prove that A implies C or that B implies C. For example, $[Q(x) \wedge P(a) \rightarrow P(x)]$ is valid if either $[Q(x) \rightarrow P(x)]$ or $[P(a) \rightarrow P(x)]$ is valid.

5. PROMOTE: $[H \rightarrow (A \rightarrow B)]$
   $[H \wedge A \rightarrow B]$.

This rule says simply that in trying to prove an implication $(A \rightarrow B)$ we can use A as an additional hypothesis.

6. BACK-CHAIN: $[H \wedge (A \rightarrow B) \rightarrow C]$
   If $[B \rightarrow C]$ is (θ)
   and $[H \rightarrow A(\theta)]$ is (λ),
   then $(\theta)(\lambda)$.

This rule applies when a term that implies the goal has an antecedent that must be proved. It says that if C can be implied from B, and $(A \rightarrow B)$, then we must try to prove A. For example, we can prove Q in $[P \wedge (P \rightarrow Q) \rightarrow Q]$ if we are able to prove P. If we instantiate H, A, B, and C in the BACK-CHAIN rule with P and Q, we obtain

If $[Q \rightarrow Q]$ is (θ)
and $[P \rightarrow P(\theta)]$ is (λ),
then $(\theta)(\lambda)$.

Obviously, $[Q \rightarrow Q]$ and $[P \rightarrow P]$ follow from the MATCH rule. In this example we have not considered substitutions.

Consider what these inference rules do and how they differ from the resolution rule. Each, with the exception of MATCH, reduces a goal to subgoals. Most of these subgoals are easily tested by MATCH; it simply tests whether there is a substitution instance for the expression. The resolution rule, by contrast, reduces clauses but does not propagate goals from one inference to the next.

IMPLY's rules are incomplete, but in most cases this does not prevent it from finding proofs of theorems. In fact, in many areas of mathematics, the great majority of proofs can be found without the extra inference rules required to make IMPLY complete. However, it can be made complete (Loveland and Stickel, 1973) and, in fact, one application warranted this (Bledsoe, Bruell, and Shostak, 1979).

Some proof procedures similar to IMPLY are described in Reiter (1976), Bibel and Schreiber (1974), Ernst (1971, 1973), and Nevins (1974, 1975).

Incorporating Heuristics into Theorem Provers

Most of the advantages derived from the use of natural-deduction theorem provers are not due to any decrease in the theoretical complexity of proofs but, rather, to the ease with which the proofs and the heuristic information incorporated into the prover can be understood. Most domain-dependent heuristics are discovered only after much analysis of attempted proofs, and the more intelligible proof structure of natural systems facilitates this analysis.

The next paragraphs describe kinds of heuristic knowledge that are typically grouped together under the heading of nonresolution theorem proving.

Reduction. The term *reduction* is used in two distinct but analogous ways. One interpretation is that reduction is the replacement of one logical expression by an equivalent, simpler expression. Alternately, reduction refers to the replacement of a term denoting an object by a simpler term. In both cases, the expression

$$L \rightarrow R$$

stands for a *reducer*. The reducer $L \rightarrow R$ is applied to a formula or term F by replacing an expression of the form $L(\theta)$ (where (θ) is a substitution) by the expression $R(\theta)$. The resulting formula or term is called an *immediate reduction*. Reductions are simpler in that they have fewer symbols or are smaller; formal requirements for simpler relations are discussed by Knuth and Bendix (1970) and Lankford (1975).

From elementary set theory, IMPLY uses (among others) the following reducers:

$$t \in (A \cap B) \rightarrow t \in A \wedge t \in B$$
$$t \in (A \cup B) \rightarrow t \in A \vee t \in B$$
$$t \subseteq (A \cap B) \rightarrow t \subseteq A \wedge t \subseteq B.$$

Examples of reducers from algebra include:

$$x + 0 \rightarrow x$$
$$x \cdot 1 \rightarrow x$$
$$x + (-x) \rightarrow 0$$
$$-(x + y) \rightarrow (-x) + (-y).$$

IMPLY maintains a list of reducers that are applied to a newly created expression until it cannot be reduced further; the resulting expression is called the *irreducible form* of the original expression relative to the list of reducers.

There are two very important properties of certain sets of reducers. A set of reducers ($\mathcal{R}$) is said to have the following:

1. *The finite termination property* (FTP). if there is no sequence of expressions $t_0, t_1, \ldots$, where t_{i+1} is an immediate reduction of t_i.
2. *The unique termination property* (UTP). if, for every expression t, all irreducible forms of t are identical.

Any set of reducers that has both the FTP and the UTP is called a ***complete set of reducers.*** There are algorithms for deciding whether a set of reducers with the FTP has the UTP (see Knuth and Bendix, 1970; Lankford 1975; Peterson and Stickel, 1977). In fact, the same algorithm can be used to extend a set of reducers that fails to have the UTP to one that does. Much research is currently being done on extending these algorithms.

Forward chaining. In addition to the rules mentioned earlier, IMPLY's set of rules includes:

FORWARD-CHAINING: $[(A \wedge (A' \rightarrow B)) \rightarrow C]$
If A is ground (i.e., has no variables) and $A' = A(\theta)$,
then $[(B(\theta) \wedge A \wedge (A' \rightarrow B)) \rightarrow C]$.

This rule differs from backward chaining in that it adds a new term to the set of hypotheses: From $(A \wedge (A' \rightarrow B))$, this rule adds $B(\theta)$ to the set of hypotheses when $A' = A(\theta)$, that is, when a substitution instance can be found for A and A'. Note that this rule does not produce smaller subgoals, as do the other rules we described, but, rather, it is used to infer auxiliary terms.

The rule contains an explicit *ground restriction* that A should have no variables. An intuitive justification for the ground restriction is that, since A is an assertion made by the hypothesis about specific objects (the ground terms) in the world, immediate consequences ($B(\theta)$) should be explored.

Many theorem provers have carried this forward-chaining rule a step further and have incorporated domain-specific knowledge into a set of *demons* that scan the hypotheses for sets of assertions. Upon finding the assertion it is looking for, a demon makes its own assertions. For example, a theorem prover might contain the following demon from elementary set theory:

> Scan the hypothesis for sets A, B, and C. If the assertions $A \subseteq B$ and $C \subseteq B$ are present, and if the set $A \cup C$ is mentioned somewhere, then assert $A \cup C \subseteq B$.

Provers using variations of this technique are described by Ballantyne and Bennett (1973), Ballantyne and Bledsoe (1977), Nevins (1975), and Hewitt (1971).

Decision procedures. Certain theories, unlike number theory, have the property that there are algorithms to decide whether a sentence is true or false in the theory. Significantly, these algorithms are often direct and can make such decisions very quickly. For example, sets of linear inequalities over the real numbers can be decided very quickly by the *simplex algorithm.* The

theory of arithmetic restricted to addition and multiplication by constants can be decided (Presburger, 1930), and, in fact, if one restricts the quantification on sentences in prenex form to universal quantification, that theory can be decided quickly (Bledsoe, 1974; Shostak, 1975). Decision procedures dealing with integration (Risch, 1969) are a main component of MACSYMA. Many fragments of theories useful in program verification have fast decision procedures (Nelson and Oppen, 1978).

A particularly interesting extension of this idea is to let the theorem prover "grow" its own decision procedures for classes of equational theories using the concept of complete sets of reducers (see Knuth and Bendix, 1970; Lankford, 1975; Huet, 1972; Lankford and Ballantyne, 1977; Ballantyne and Lankford, 1979; Peterson and Stickel, 1977).

Induction. Induction is another area in which the addition of heuristics can improve the performance of a prover. Since the development of a sophisticated set of such heuristics is one of the major achievements of the Boyer-Moore theorem prover, we refer the reader to Article XII.D.

Examples and counterexamples. Examples and counterexamples play an important but poorly understood role in automatic theorem proving. Specifically, if T is a set of axioms for a theory and if $H \rightarrow C$ is an attempted theorem, then an example is an interpretation of the predicate, function, and constant symbols that satisfies H and the axioms.

For example, let T be the axioms for the real numbers, and let H be $[f(a) \leq 0 \wedge f(b) \geq 0 \wedge \text{CONTINUOUS}(f, a, b)]$, where f, a, and b are constants and $\text{CONTINUOUS}(f, a, b)$ means that the function f is continuous on the closed interval $[a, b]$. Then the assignment

$$a \leftarrow 0$$
$$b \leftarrow 1$$
$$f \leftarrow ((\lambda)x)(2x - 1)$$

is an example.

To see how this example might be useful in controlling the search for a proof, suppose that the theorem prover is asked to prove the conclusion $C = (\text{SOME } x)(f(x) = 0)$, given the above axioms and hypotheses. Suppose that, in the course of proving C, the prover encounters the subgoal $f(t) \leq 0$, where t is a term that evaluates to 3/4 in the example. Since $f(t) = f(3/4) = 2 \cdot 3/4 - 1 = 1/2$ and since 1/2 is not less than or equal to 0, the prover is allowed to discard this subgoal. Several theorem provers have incorporated examples as a *subgoal filter* (Gelernter, 1959; Reiter, 1976; Bledsoe and Ballantyne, 1979). In all these provers, the examples must be generated by the user. However, Bledsoe and Ballantyne describe a program that, when given an example, extends the interpretation to include the skolem functions and constants that result from quantifier elimination.

It seems likely that mathematicians use examples much more often as subgoal *proposers* than as subgoal *rejectors*. Mathematicians often use examples

to guide the search for a proof from beginning to end. Since they usually discover theorems by building and inspecting examples, it seems likely that the same examples would be useful in proving these theorems. Constructing good examples is a very difficult task but one that must be understood if reasonably competent theorem proving is to be done by computer. Lenat's AM system (1976; Article XIV.D4c) constructed and used examples to help make conjectures.

Conclusion

Nonresolution, or natural-deduction, proof procedures are designed to develop proofs in a goal-directed manner that is easy for humans to understand. Unlike resolution methods, natural deduction uses many proof rules to reduce goals to subgoals. In addition, natural-deduction systems often include domain-specific heuristics to speed up parts of a proof.

Any proof that can be derived by natural deduction can also be derived by resolution, given enough time. The advantage of natural deduction is chiefly that the proofs it produces are relatively easy to understand. This is very important whenever there is interaction between an automatic theorem prover and a human.

References

The IMPLY system is discussed in Bledsoe and Tyson (1975).

D. THE BOYER–MOORE THEOREM PROVER

THE Boyer-Moore Theorem Prover (BMTP; Boyer and Moore, 1979) embodies an extensible mathematical theory (recursive function theory) in which theorems can be stated and automatically proved. The system is designed to prove theorems by continuously rewriting the current formula (Bledsoe, 1971, 1977) without ever having to backtrack and alter a decision. While each rewriting rule is sound, formal equivalence is not necessarily preserved; thus, the system is not complete. But heuristics are employed to guide the rewriting process, applying rules that the system believes will allow retention of the "theoremness" of a formula. The theory can be extended by new function definitions and new data types. Novel features include the automatic use of structural induction (Burstall, 1969) and recursive quantification (Skolem, 1967). The relations between recursion, termination, and the inductively defined data objects allow the BMTP to produce induction proofs automatically. Recursive functions, used as an alternative to quantification, offer a powerful form of expression when dealing with finitely constructed objects such as the discrete mathematical structures employed by computer programs.

Rather than operate in the predicate calculus (see Article III.C1, in Vol. I), the Boyer-Moore Theorem Prover treats axioms and theorems as functions. Axioms have the values non-F (true) or F (false). A theorem is proved by showing that the value of its function is non-F. For example, a statement that multiplication is distributive over addition would have appeared in QA3 (Green, 1969; see also Article III.C1, in Vol. I) as:

```
FORALL x FORALL y FORALL z  SUM(y,z,a1) AND PRODUCT(x,a1,a) AND
         PRODUCT(x,y,b1) AND PRODUCT(x,z,b2) AND SUM(b1,b2,b) AND
         EQUAL(a,b)
```

(where x, y, z, a, a_1, b, b_1, b_2 are all variables). In the BMTP, the theorem becomes:

```
(EQUAL (TIMES x (ADD y z)) (ADD (TIMES x y) (TIMES x z))) .
```

The Boyer-Moore Theorem Prover automatically proves the theorems it is presented with, possibly using rewrite lemmas that have been retained from the proofs of previous theorems or axioms that have been added by the introduction of new data types. Most theorems cannot be proved from first principles, so the user must structure the proof by determining intuitively which lemmas will be necessary. These are then proved as theorems in their own right and saved. Since lemmas must be proved before they can be

automatically used, the BMTP is assured of the validity of the proof of the final theorem. Even theorems that can be proved without lemmas can have their proofs speeded up by the use of lemmas. If the BMTP fails to prove the desired result, the proof attempt helps the user determine where the proof went awry and formulate new lemmas. Thus, the BMTP is an automatic theorem prover in the sense that the user specifies only what to prove, not how to prove it. But if a proof fails, the user provides a bit of the "how" by formulating an appropriate lemma.

The system is experimental and is continually being tested and improved. It has proved approximately 400 theorems, including the soundness and completeness of a tautology checker for propositional calculus, the equivalence of interpreted and optimized compiled code for a simple arithmetic language, the correctness of the Boyer-Moore fast string-searching algorithm, and the prime-factorization theorem.

The Theory

The syntax of the theory is closely related to the prefix notation in LISP. Terms are variables or are specified by $(f\ x_1 \ldots x_n)$, where f is an n-ary function symbol and all x_i are terms. Constants are represented as 0-ary functions (e.g., (TRUE), (FALSE), (ZERO)). The variables in any formula are implicitly universally quantified.

Functions are introduced by adding the equality axiom:

$$(f\ x_1 \ldots x_n) = \text{(function body)}\ .$$

To retain consistency, the BMTP requires that each newly defined function be either nonrecursive or recursive but provably total. The proof of totality is based on the notion of measure functions and well-founded relations. This is discussed in detail later in this article in the section on induction.

In making function definitions it is often necessary to include tests that allow the returned value of a function to be one of a set of terms. The usual treatment of logic does not allow for the embedding of propositions within terms, so the BMTP recreates the effects of propositions at the term level. Boyer and Moore create four axioms to define the functions EQUAL and IF; these form the core of the BMTP. We abbreviate (TRUE) as T and (FALSE) as F, and add the axiom that T and F are distinct:

1. T ≠ F
2. X = Y ⇒ (EQUAL X Y) = T
3. X ≠ Y ⇒ (EQUAL X Y) = F
4. X = F ⇒ (IF X Y Z) = Z
5. X ≠ F ⇒ (IF X Y Z) = Y

(For those readers who are not familiar with LISP notation, `(IF X Y Z)` means *If X, then Y; else Z.*) Thus, the term `(IF X Y Z)` has the value `Z` if the proposition `X = F` is true and it has the value `Y` if `X = F` is false.

Boyer and Moore do not define predicates but, instead, deal within a theory of functions. Proving that the value of a function is not `F` is the way the BMTP proves that a function is a theorem. Functional versions of common logical connectives are defined with `IF`. These definitions capture the semantics of the common logical connectives:

1. `(NOT P) = (IF P F T)`
2. `(AND P Q) = (IF P (IF Q T F) F)`
3. `(OR P Q) = (IF P T (IF Q T F))`
4. `(IMPLIES P Q) = (IF P (IF Q T F) T)`

In addition to these and other functions, the BMTP allows the creation of arbitrary data types. These are typically defined inductively and made known to the system by the Shell mechanism (discussed below), which adds axioms that are guaranteed to leave the theory consistent. Data objects are considered to be finitely constructed. Data types are mutually exclusive yet not assumed to be exhaustive. This guarantees that the subsequent addition of new data types will not invalidate previously proved theorems.

Proofs within the BMTP are accomplished by absorption, idempotency, the law of excluded middle (e.g., $T \vee X \rightarrow T$, $F \vee X \rightarrow X$, $X \vee \neg X \rightarrow T$, and their commutative counterparts), and induction principles. Recursion as a control structure is analogous to inductively defined data types as a data structure. The proof-theoretic counterpart of these two is the Generalized Principle of Induction, or Noetherian Induction. A consistent induction mechanism is presented within the theory. It allows a base case as well as k remaining induction steps, each of which can contain several induction hypotheses. It requires a relation that is well-founded on a measured set of variables over all substitutions required to instantiate the $k + 1$ cases. Heuristic methods are employed in the BMTP to formulate this schema; they are discussed later in this article in the section on induction. A well-founded relation r is one that admits no infinitely decreasing sequences. That is, there cannot exist an infinite sequence $1, 2, \ldots$ such that $(rX_{i+1}X_i)$. A simple well-founded relation is $<$ on the nonnegative integers, since for any X_1 we cannot find an infinite sequence of x_i such that

$$\cdots X_{i+1} < X_i < X_{i-1} < \cdots < X_1 .$$

The Shell mechanism. The Shell mechanism is used to introduce new data types. It is just a syntactic form from which consistent and complete type-axioms are created. As an illustration, the definition of lists by the Shell mechanism is as follows:

```
add the shell CONS, of 2 arguments
recognizer LISTP
accessors CAR, CDR
default values "NIL", "NIL" .
```

A few of the important axioms that were added (with symmetric CDR axioms) are the following:

`(LISTP (CONS x y))`	— a CONS of two things is always a list
`(EQUAL (CAR (CONS x y) x))`	— definition of the CAR accessing function
`(IMPLIES (LISTP x) (LESSP (CAR x) x))`	— a measure property used in proving termination
`(EQUAL (EQUAL (CONS a b) (CONS x y))` `(AND (EQUAL a x) (EQUAL b y)))`	— two CONSes are equal if their parts are equal

`(IMPLIES (LISTP x)`
`(EQUAL (CONS (CAR x) (CDR x))`
`x))`
	— the system can trade CARs and CDRs for CONSes

Overview of the Theorem Prover

The BMTP proves that a formula is a theorem by continually rewriting the formula until it is reduced to T. The BMTP operates in a strictly linear manner without backtracking. This strategy leads to a stratification of the classes of rewrite rules, so that the more conservative transformations (i.e., those which guarantee equivalence) are attempted first. Induction rewrite rules are applied last, since they are the least conservative transformations and it is important that induction be applied to the simplest and most general form of a formula. As a consequence, many of the rewrite rules have been designed to produce a formula that is more amenable to inductive arguments. We will now discuss these rule classes. Rules at level $i + 1$ are tried only when all rules at level i fail to be applicable. If a rewrite rule applies at any level of the hierarchy, the formula is rewritten and the entire theorem prover is recursively invoked on the new formula.

Simplification

The formula is rewritten by the logical proof rules, the initial axioms, the axioms added by function and data-type definitions, and retained lemmas that were previously proved as theorems. (The formula is also rewritten to conjunctive normal form, or *clause form;* see Article XII.B.) All these rewriting rules retain truth-value equivalence. The *Simplifier* is a small theorem-prover in its own right. Examples of the information known to the Simplifier are:

1. Logical Proof Rule:
   `X ∨ T = T`

2. Initial Axiom:
   `x = y ⇒ (IF x y z) = y`
3. Function Axiom:
   `(APPEND x y) = (IF (LISTP x)`
   `(CONS (CAR x) (APPEND (CDR x) y)) y)`
4. Data-type Axiom:
   `(CDR (CONS x y)) = y`
5. Lemma:
   `(APPEND (APPEND x y) z) = (APPEND x (APPEND y z))`

Simplification is sufficient to prove the following formula (which is the base case of the induction needed to prove that APPEND is associative):

```
(IMPLIES (NOT (LISTP A))
         (EQUAL (APPEND (APPEND A B) C)
                (APPEND A (APPEND B C)))) .
```

Knowing that A is not a list allows the APPEND functions to open up and return their second arguments: see the functional definition of APPEND above. The formula simplifies to:

```
(IMPLIES (NOT (LISTP A))
         (EQUAL (APPEND B C)
                (APPEND B C))) .
```

Since the two APPEND terms are identical, this simplifies to:

```
(IMPLIES (NOT (LISTP A)) T) .
```

This in turn simplifies to T, since the formula is equivalent to the clause (LISTP A) ∨ T, which by the above proof rule is rewritten to T.

If simplification cannot determine the truth value of a formula, it will probably be necessary to apply the induction rewriting rules. The next four cases illustrate how the formula is prepared for induction.

Elimination of Undesirable Concepts

The BMTP restates a formula, trading some functions for others when the substituted formulas are easier to rewrite or have more lemmas involving them. This type of rule is a special subclass of the general simplification rules and is handled separately since it requires special processing. An example of this kind of rule is:

`(p x) = (p (CONS A B))`, if x is known to be a list.

An example of its application is found in the proof of the theorem that the function REVERSE is its own inverse:

```
(IMPLIES
  (AND (LISTP X)
       (EQUAL (REVERSE (REVERSE (CDR X))) (CDR X))
       (PLISTP (CDR X)))
  (EQUAL (REVERSE (APPEND (REVERSE (CDR X))
                          (CONS (CAR X) "NIL")))
         X))

= (IMPLIES
    (AND (LISTP (CONS A B))
         (EQUAL (REVERSE (REVERSE (CDR (CONS A B)))) (CDR (CONS A B)))
         (PLISTP (CDR (CONS A B))))
    (EQUAL (REVERSE (APPEND (REVERSE B)
                            (CONS A "NIL")))
           (CONS A B))) .
```

Here we have traded a CAR and CDR for a CONS. Note that this transformation was applicable since X was known to be a list from the hypothesis of the implication. A and B are new variable names.

This fairly complicated formula is passed back to the Simplifier, which rewrites it as:

```
(IMPLIES
  (AND (EQUAL (REVERSE (REVERSE B)) B)
       (PLISTP B))
  (EQUAL (REVERSE (APPEND (REVERSE B)
                          (CONS A "NIL")))
         (CONS A B))) .
```

Use of Equalities

The BMTP uses equalities by substituting equals for equals, and then it usually removes the equality term from the formula. This is not guaranteed to be complete, but the heuristic decision procedure in BMTP that decides which terms to substitute performs excellently. The equality term is removed to simplify the statement of the formula (which hopefully is still a theorem). Two distinct classes of substitutions—uniform substitution and cross-fertilization—are performed.

Uniform substitution. If the term (EQUAL x ev) is found, where x is a term and ev is an explicit value, then ev is uniformly substituted for x within the rest of the formula. The symmetric case applies.

Cross-fertilization. If the term (EQUAL x y) is found, where both x and y are not explicit values, and another term of the form "(p ⟨any term⟩ ⟨term that contains y⟩)" is found, then x is substituted for y only in the right-hand side of p, and the equality is removed from the formula. The symmetric case applies. This heuristic is closely related to the way induction is performed; it is

designed to allow maximum use of the induction hypothesis. The connection is a bit subtle and the reader is referred to Boyer and Moore's (1979) description.

Continuing the above example, the antecedent has an equality of the form "(EQUAL x B)" and the consequent term is of the form "(p (term) (term with B))." so we cross-fertilize. This results in:

```
(IMPLIES
  (PLISTP B)
  (EQUAL (REVERSE (APPEND (REVERSE B)
                          (CONS A "NIL")))
         (CONS A (REVERSE (REVERSE B))))) .
```

Generalization

A further simplification can be accomplished by replacing a term in the formula by a variable, thus generalizing the formula and allowing an induction on the new variable position in the formula. Hopefully, by the time we reach this point, the internal structure of the term has already contributed its significance to the proof and can be ignored. To prevent the formula from becoming overgeneralized, the BMTP can add certain type-restrictions to the variable introduced. The REVERSE example that we have been following does not adequately illustrate generalization, so we move temporarily to a different example:

```
(EQUAL (APPEND (FLATTEN Z)
               (APPEND (FLATTEN V) ANS))
       (APPEND (APPEND (FLATTEN Z) (FLATTEN V))
               ANS))

= (IMPLIES (AND (LISTP A) (LISTP B))
           (EQUAL (APPEND A (APPEND B ANS))
                  (APPEND (APPEND A B) ANS))) .
```

Here, (FLATTEN Z) and (FLATTEN V) have been generalized to A and B, respectively. Type information has been added showing that both A and B are list data types, since the system is aware of a theorem stating that FLATTEN always produces a list. The formula now is just the statement that APPEND is associative.

Elimination of Irrelevant Terms

In performing the above transformations, it is often the case that irrelevant terms are left in a formula. Removing these terms cleans up the formula. While these terms are difficult to spot in general, there are two special cases, shown as rules 1 and 2 below, that frequently occur. In both cases, all the

terms of a formula are first partitioned into equivalence classes with term 1 in the same class as term 2 if they share a common variable.

Rule 1. If a class contains only nonrecursive functions, then all terms in the class are removed from the formula. If these formulas were always non-F, the Simplifier should have been able to prove this fact. Passing these terms on to the Induction mechanism will not help, since the terms are not recursively defined.

Rule 2. If a class contains a single recursive function, it is removed. A single function that cannot be shown to be always non-F by the Simplifier probably can assume non-F values.

Continuing our example of the proof `(EQUAL (REVERSE (REVERSE X)) X)`, the theorem is generalized to:

```
(IMPLIES
  (PLISTP B)
  (EQUAL (REVERSE (APPEND X (CONS A "NIL")))
         (CONS A (REVERSE X))))
```

by replacing all occurrences of `(REVERSE B)` with `X`. No extra type information is added during generalization. The antecedent is eliminated by rule 2, leaving the formula:

```
(EQUAL (REVERSE (APPEND X (CONS A "NIL")))
       (CONS A (REVERSE X))) ,
```

which is a statement asserting that reversing the concatenation of `X` and `A` is equivalent to concatenating `A` with the reverse of `X`.

Performing an Induction

If, in the course of these rewrites, the theorem has still not been reduced to T, the BMTP automatically formulates a valid induction argument to try to prove the theorem. The heuristics employed here represent the heart of the BMTP. Inductions are formulated by using information collected at the time the function is defined and at the time the actual induction is needed.

Function-definition time. When a function is defined, the system must prove that the function terminates before allowing the definition. Termination is proved by finding a well-founded function that decreases when applied to a subset (measured set) of the arguments used in all recursive calls. The system exhaustively searches through all lexicographic orders of all well-founded functions (LESSP is initially the only one, but others are added by the Shell mechanism) applied to all subsets and permutations of a function's arguments. These are all collected in a set of induction templates that are associated with

the newly defined function. These templates include the form of the induction to be performed and all of the variable substitutions that will need to be made.

The following illustrates the creation of induction templates at function-definition time for REVERSE, which is defined as:

```
(REVERSE X) = (IF (LISTP X)
                  (APPEND (REVERSE (CDR X)) (CONS (CAR X) "NIL"))
                  (CONS X "NIL")) .
```

The proof of termination is fairly simple, since REVERSE is monadic and there is only one recursive function call within its body. The BMTP utilizes the information that the recursive function call is executed only if X is known to be a list. Thus, to prove that REVERSE terminates, it tries to prove the theorem:

```
(IMPLIES (LISTP X) (LESSP (CDR X) X)) .
```

The system proves this theorem (it recursively calls itself) by noticing that this formula is equivalent to an axiom added by the Shell mechanism during the definition of lists. This is the only way the system can prove termination, so the only induction template produced is:

```
(AND (IMPLIES (NOT (LISTP x)) (p x))
     (IMPLIES (AND (LISTP x)
                   (p (CDR x)))
              (p x))) .
```

This states that, to prove the formula (p x) where p involves the REVERSE function, it is sufficient for the BMTP to prove that:

1. If x is not a list (the base case), then (p x) can be proved.
2. If x is a list and (p (CDR x)) is assumed to be true (the induction hypothesis), then (p x) can be proved.

Typically, the formula p will also involve other recursive functions that have their own induction templates. The problem of which induction template to use cannot be handled at function-definition time (since the BMTP has no way to determine how a newly defined function will be used) and is handled when the induction rewrite rules are trying to rewrite the formula.

Instantiation time. When an induction rewrite rule is attempted, the induction templates for all recursive functions in the formula are retrieved. These templates are then sifted by the following rules:

1. Only legal templates (with valid substitution instances) are retained. Substitutions may be invalid for many reasons, the most common that

the template requires that a nonvariable argument be used as an induction variable. The REVERSE induction template could not be used if the formula p involved only terms like (REVERSE (f x)): hopefully, the generalization heuristics will substitute a variable for the function (f x).

2. Induction schemata are obtained when the legal templates are instantiated by performing the required substitutions. All subsumed induction schemata are discarded. This means that the system will discard weaker induction arguments for ones with a richer case structure (duplicates are removed by this method also).

3. The remaining templates are then merged. Two templates are merged if they contain a common induction variable, allowing for the final induction scheme to contain induction hypotheses for every relevant induction variable. Thus, if one induction scheme requires induction on the variables x and y and another requires induction on the variables y and z, it seems plausible to require simultaneous induction on all of x, y, and z.

4. If more than one scheme still exists and there is one "unflawed" scheme, then all "flawed" schemes are discarded. An induction scheme is unflawed if every occurrence of an induction variable is in a position where it is decomposed.

5. Finally, if more than one scheme still exists, a scoring function determines which one to use.

6. The final scheme is then instantiated for the specific formula to be proved.

Boyer and Moore (1979) report that 90% of all inductions' arguments yield only one unflawed scheme and, of the remaining 10%, half have no unique correct scheme (i.e., the theorems are symmetric in some variables).

Continuing the REVERSE example, the BMTP is about to create an induction argument for proving:

```
(EQUAL (REVERSE (APPEND X (CONS A "NIL")))
       (CONS A (REVERSE X))) .
```

It determines the induction schemata for REVERSE and APPEND, and since both functions perform CDR recursions on X, their schemata are merged to create the unique induction schema, which is finally used:

```
(AND (IMPLIES (NOT (LISTP X)) (p X A))
     (IMPLIES (AND (LISTP X)
                   (p (CDR X) A))
              (p X A))) .
```

Themes of the Boyer-Moore Theorem Prover

Proof by induction. The outstanding feature of the BMTP is that it automates induction proofs. Since most common data-types (integers, lists, trees, formulas) are defined inductively, it is imperative that theorem provers that prove properties of programs have the capability of performing inductive arguments (automatically or manually). The excellent performance of the BMTP is in a large part due to the heuristic methods employed in constructing induction proofs. These heuristics form the core contribution the BMTP has made to AI research.

Referencing problem. A key problem in current theorem-proving systems is the performance degradation due to increased knowledge. While increased knowledge should improve a system's performance, it typically just expands the possible solution space, causing excess searching. This has been named the *referencing problem* by Bledsoe (1974). Resolution theorem-provers suffer greatly from this problem. Such methods as proof by analogy (Kling, 1971) have been used to restrict the reference set, but they have met with little success. The BMTP does not address this issue with any more sophistication than trying the rewrite rules in reverse chronological order (with complex results first). This simple strategy has proved effective even when operating within an environment that contains approximately 400 theorems.

The language of the theorem prover. Since the main application of the BMTP has been to prove properties of programs, a possible misconception should be avoided. There is a difference between the language used to express formal statements whose validity is being proved and the language used to express a program. The theory is just a mathematical tool for making precise assertions about the properties of discrete mathematical objects. The language used to express the theory is closely related to the pure LISP programming language and should be considered as an alternative to the use of the predicate calculus. Frequently, programs can be written as functions within the theory (since the semantics of a LISP-like program can be easily captured within the language of the theory) just as it is possible to use predicate calculus as a programming language (Kowalski, 1974). But a distinction should be made between the language used to express theorems and the programming language used to describe an algorithm about which the BMTP is proving theorems. When proving properties about programs, the user applies a relevant theory of program semantics to derive formal statements whose validity implies that the program has the desired properties. These statements are then translated into the theory on which the BMTP operates. The BMTP can then be instructed to try to establish the validity of these statements. To illustrate this fact, the proof of the correctness of the compiler is expressed by McCarthy's functional method, while the correctness of the string-searching algorithm is expressed by Floyd's method of inductive assertions.

Performance. Two performance measures are relevant to theorem provers. The first is the system's ability to represent typical facts and theorems in the domain of interest (epistemological adequacy). The second is the ability to prove theorems within a reasonable amount of time. Both performance measures contain ambiguity (e.g., "typical," "reasonable"). But in the BMTP, many interesting facts and theorems can be represented, and proof times are commensurate with a user's patience when debugging proofs interactively. The BMTP has been applied to a large number of theorem-proving tasks, some of which are very difficult by human standards. Most theorems are proved in well under a minute, although most proofs require lemmas to be proved previously. Nevertheless, this is one of the most powerful theorem provers available.

References

Boyer and Moore discuss their theorem prover in their 1979 article.

E. NONMONOTONIC LOGICS

SEVERAL FORMS of nondeductive reasoning have attracted careful scrutiny. Purely deductive reasoning techniques have long been recognized as inadequate for capturing all intelligent thought. Statistical and inductive reasoning, which concern inexact and generalizing reasoning, have received much study as possible extensions or alternatives to deductive reasoning. Nonmonotonic reasoning, recently formalized in nonmonotonic logics, is the latest extension to deductive reasoning. This article sketches the nature of, reasons for, and approaches to nonmonotonic logics.

The Task of Logic

The task of logic is the judgment of arguments. Historically, logic has been the science of argumentation, the study of which arguments are good and which are not good. Different purposes engendered different conceptions of good. Arguments to convince capricious, distracted, and sometimes irrational humans were judged by the standards of effective rhetoric, which concern, among other things, the size, structure, motivation, and emotional impact of arguments and their steps. Inductive logics judged arguments that made generalizations; statistical logics judged arguments that dealt with frequencies and probabilities; and deductive logics judged arguments that made restatements, that is, truth-preserving inferences.

While important insights were gained into the philosophical and practical questions underlying rhetorical, statistical, and inductive reasoning, perhaps the philosophically most striking advances were made in connection with deductive reasoning. Philosophers, logicians, and mathematicians explored the powerful ideas of formal languages, truth-theoretic semantics, set theory, and the mathematics of formal systems, model theory, and proof theory. These ideas proved so fruitful that logic for the most part came to be identified with deductive logic, the study of truth-preserving inferences. This identification grew so strong that many of the proposed nondeductive logics have been attacked as false logics. But logic is a science of thought and argument, not merely a science of truth-preserving inferences.

The Task of Nonmonotonic Logic

The task of nonmonotonic logics is to judge cases of nonmonotonic reasoning, that is, reasoning that involves adopting assumptions that may have to be abandoned in light of new information. For example, a scheduling secretary

may employ the inference rule that he (or she) should schedule each new meeting on the closest future Wednesday unless and until he finds reasons for scheduling the meeting otherwise. While working out the week's schedule, the secretary may tentatively schedule the first meeting on the next Wednesday, only to reschedule it later, thereby abandoning his initial assumption, when he learns that a meeting is requested for that Wednesday specifically to accommodate a visitor.

This reasoning is called nonmonotonic in contrast to the monotonicity of the set of theorems of a set of axioms in deductive logic. In deductive logic, the addition of new axioms to a set of axioms can never decrease the set of theorems. At most, the new axioms can give rise to new theorems, so that the set of theorems grows monotonically with the set of axioms. In nonmonotonic logics, the set of theorems may lose members as well as gain members when new axioms are added.

Reasoning by Default

Two cases of nonmonotonic reasoning have been studied: reasoning by default and reasoning by circumscription.

The defaults of reasoning by default are statements or rules according to which (as in the scheduling example above) some statement is to be believed, unless and until otherwise demonstrated. Defaults can be found in many places in standard AI techniques. They are used in stating generalities to which exceptions may be acknowledged without catastrophe. For example, a default might be that all birds can fly; penguins and ostriches are exceptions. In structured knowledge-representation systems (see Article III.C7, in Vol. I), such defaults often take the form of default fillers of frame slots. For example, an airline reservation system might describe each customer with a *passenger* frame in which the *class* slot has the default value *coach*. Defaults also enter into many knowledge-representation systems implicitly through what is known as the *closed-world assumption*. The closed-world assumption is that all relationships not explicitly stated to hold do not hold. For example, typical procedures for inheriting statements in one frame from more general frames by way of *generalization* links assume that a frame is generalized only by those frames explicitly listed as generalizations or, in turn, by their generalizations. Thus, if the *elephant* frame has a sole generalization link to the *mammal* frame, the inheritance procedures will search only *mammal* and not any other frames, in spite of the possibility that new generalization links may be attached to *elephant* later and would then be searched as well. Yet another use of defaults is in the typical *STRIPS assumption* that performed actions change none of the program's beliefs about the world except those explicitly listed in the description of the action (see Article XV.B). For example, a description of a robot's action of moving from one location to another would list only changes

in beliefs about the robot's position. When the robot moves, the STRIPS assumption default would leave its belief about world geography intact.

Reasoning by Circumscription

Another case of nonmonotonic reasoning, which may well overlap defaults in some (or even all) cases, is that of parsimonious reasoning, or reasoning by circumscription. In reasoning about some problem, one often assumes that the problem involves only those objects and relationships that it mentions, and no others. The inheritance procedures mentioned above made such an assumption (the closed-world assumption) about the nonexistence of unlisted generalization links and generalizing frames. As another example, in the well-known missionaries-and-cannibals problem of traversing a river uneaten, one typically does not think of solutions involving bridges, rocket ships, handcuffs, murder of the cannibals, or holes in the boat. Another way of viewing the circumscription principle is the assumption that all qualifications to the problem have been stated explicitly.

Formal Characterizations of Defaults

Two sorts of detailed formalizations of nonmonotonic defaults have been proposed, namely, Reiter's logic of defaults and McDermott and Doyle's nonmonotonic logics.

Both logics roughly interpret *Default S as S is provable unless and until S can be disproved.* The difficulty with this interpretation is its circularity, that what can be inferred depends on what inference rules are applicable, while, at the same time, what inference rules are applicable depends on what can be inferred. For example, suppose that we decide to use only the ordinary logical rules of inference in attempting to disprove statements and that the information to be captured consists of three statements: Default A, Default B, and $\neg(A \wedge B)$. Here, neither A nor B can be disproved using the ordinary logical rules of inference, so we declare both A and B to be provable by means of the default statements. These two new conclusions are inconsistent with $\neg(A \wedge B)$. Instead of declaring the initial three statements to be inconsistent, the nonmonotonic logics try to refine the notions of provability to say that there are two coherent interpretations of these axioms, namely, one in which A and $\neg B$ are provable and one in which B and $\neg A$ are provable. This is a big departure from ordinary logic, in which a single set of axioms has exactly one set of conclusions that can be drawn from it. The key problem addressed by the nonmonotonic logics is that of providing some well-defined semantics for defaults that allows a single set of axioms and defaults to have several coherent interpretations.

In all the nonmonotonic logics, the meanings of *provable* and *consistent* for a statement and a set of axioms are defined nonconstructively by a

mathematical definition of what *coherent* sets of conclusions are, relative to a given set of axioms and defaults. These definitions are nonconstructive primarily because the coherent interpretations supplied by the logics are in general not even recursively enumerable. Roughly put, the logics declare that interpretations are found by adding in as many statements (assumptions) as possible, in accordance with the defaults, but at the same time avoiding adding in so many assumptions as to produce an ordinary logical inconsistency. In the above example, for instance, the two coherent interpretations of the three statements are produced by adding in just one of the assumptions, A or B. By the time one assumption is added in, the negation of the other can be deduced by ordinary logical rules of inference, so that the other assumption is ruled out, as it would lead to an inconsistency. This rough description of the semantics provided by the logics does not do them justice. For the precise definitions involved, the reader is referred to the original papers (Reiter, 1980; McDermott and Doyle, 1980).

While Reiter's and McDermott and Doyle's approaches to formalizing defaults have much in common in the way they interpret defaults and in their major theoretical properties, they differ in logical form, as one approach formalizes defaults as inference rules and the other as modal formulas. Unless one is vitally interested in logic for its own sake, or in pursuing the future development of better nonmonotonic logics, these differences in logical form can be passed over as small differences in notation for capturing the same ideas.

Reiter (1980) formalizes defaults by adjoining a new sort of inference rule called a default to an ordinary logic of statements and inference rules. Default inference rules are of the form *If P, and it is consistent to assume Q, then infer R*, written $P : Q/R$, where P, Q, and R are ordinary formulas. Given condition P, a default allows the inference of R providing that Q is not disprovable. With this notation, the simplest sort of default, that of *Assume A if it cannot be disproved*, is written simply as "$: A/A$": that is, P is empty and $Q = R = A$.

Instead of stating defaults as inference rules, McDermott and Doyle (1980; McDermott, 1980) state defaults as modal formulas. They use an ordinary logical language extended by the unary modal operator *not-disprovable*. The analogue in nonmonotonic logic of a default inference rule $P : Q/R$ of the logic of defaults is $P \wedge$ *not-disprovable* $Q \rightarrow R$. Thus the simplest sort of default is stated in these nonmonotonic logics as *not-disprovable* $A \rightarrow A$. Although we said earlier that nonmonotonic logics and the logic of defaults are for many purposes syntactic variants, that is not really true. The modal nonmonotonic logic formulations are, for better or worse, actually more expressive than the nonmodal logic of defaults. This is because one can make statements about defaults: for example

$$\textit{not-disprovable}(\textit{not-disprovable}\ A \rightarrow A) \rightarrow (\textit{not-disprovable}\ A \rightarrow A),$$

in nonmonotonic logics, whereas in the logic of defaults no means exists for referring to the default inference rules.

The Genesis of Practical Nonmonotonic Inference Rules

Neither of these approaches says anything about which nonmonotonic statements or rules should be used in representing information about a particular domain. The logics all leave that decision to the AI system designer. However, McCarthy (1980) and Dacey (1978) have each developed theories that appear to bear on the problem of formulating defaults. McCarthy formalizes reasoning by circumscription as an explicitly nonmonotonic rule of inference. Dacey, on the other hand, formalizes his *theory of conclusions* in terms of classical decision theory, rather than in terms of nonmonotonic reasoning.

The idea of circumscription, in McCarthy's (1980) treatment, becomes an inference rule for formulating sets of assumptions on the basis of the available information. The circumscription inference rule computes axiom schemata from sets of axioms, schemata that can be applied to make a variety of assumptions. To circumscribe a set of axioms A with respect to some predicate P mentioned in A, one constructs a sentence schema stating that the only objects satisfying P are those whose doing so follows from the axioms A. All statements following via ordinary deductive rules of inference from that sentence schema are said to be the conclusions reached by *circumscriptive inference* with respect to P from the original axioms A. For example, suppose we know only one red-haired person, our friend Jane. If we see someone looking like Jane in the crude sense of merely being red-haired, we might, by circumscription, assume that that person is Jane, because Jane is the only person we know fitting that description. This inference is nonmonotonic, of course, since if we now learn that Jane has an identical twin sister Joan, we can no longer conclude that anyone who looks like Jane is Jane. Expressed formally in terms of McCarthy's circumscription, this example might be translated as follows. We start with the set of axioms $A = (\textit{red-haired}(\textit{Jane}))$ and circumscribe on the predicate *red-haired*. The circumscription of this predicate in A is the axiom schema

$$\phi(\textit{Jane}) \wedge \forall x\,(\phi(x) \rightarrow \textit{red-haired}(x)) \rightarrow \forall x\,(\textit{red-haired}(x) \rightarrow \phi(x)).$$

If we now substitute our only known instance of a red-haired person into this schema, that is, if we substitute the formula $x = \textit{Jane}$ for $\phi(x)$, we get

$$\textit{Jane} = \textit{Jane} \wedge \forall x\,(x = \textit{Jane} \rightarrow \textit{red-haired}(x)) \rightarrow \forall x\,(\textit{red-haired}(x) \rightarrow x = \textit{Jane}).$$

The first two parts of this formula are true, and simplifying it leaves the resulting assumption, or *default*, $\forall x\,(\textit{red-haired}(x) \rightarrow x = \textit{Jane})$, which we can apply to any new person who looks like Jane (i.e., is red-haired). Yet this inference is nonmonotonic in that, if we add the new axiom $\textit{red-haired}(\textit{Joan})$

to A, we can no longer draw any such identifying conclusion. At best, we can infer by another application of circumscription the less specific conclusion $\forall x\,(red\text{-}haired(x) \rightarrow x = Jane \vee x = Joan)$.

Another approach to forming and rejecting tentative hypotheses, the *theory of conclusions* developed by Dacey (1978) after a suggestion of Tukey (1960), can be viewed as proposing a general rule about when to adopt and when to abandon defaults. Dacey formulates conclusion theory in terms of classical decision theory rather than in the proof-theoretic terms of the preceding approaches. Classical decision theory analyzes how the strength of each of one's hypotheses about the world should be revised with each new evidential fact. The intent of conclusion theory is to avoid the continual reevaluation of all hypotheses, to instead accept certain strong hypotheses as *conclusions*, and to hold these conclusions unless and until the introduction of very strong contrary evidence. Although Dacey apparently intends that the set of conclusions be the set of beliefs of the reasoner, his reasoner is isolated and unreflective, in that the rules of adoption and abandonment are used in developing scientific laws *de novo*. Once communication or summaries of conclusions are desired, as in writing an initially substantive AI program, the form of each conclusion seems to approximate that of a default. Thus, conclusion theory might be adapted to the role of judging the propriety of adopting or abandoning defaults.

The Mathematics of Theory Evolution

Each of the approaches above treats in detail primarily the atoms of reasoning, either individual inference steps or the sets of beliefs preceding and following the inference step. So far, much less attention has been devoted to classifying the larger, more complex ways in which nondeductive inferences can change the current set of beliefs of a reasoner. The beginnings of a larger analysis of theory evolution are touched on by McDermott and Doyle (1980), Doyle (1979, 1980), Gumb (1978, 1979), Weyhrauch (1980), and, less formally, in the philosophy of science literature in general (e.g., Quine and Ullian, 1978).

References

The area of nonmonotonic reasoning has to date supplied more questions than definitive answers, but the questions it raises are vital. For further information, peruse the papers collected in the special issue of *Artificial Intelligence* on nonmonotonic logic (such as those cited above) and the papers indexed in the bibliography by Doyle and London (1980). Further fruitful formalizations of nondeductive reasoning techniques may well be awaiting discovery. For example, Collins (1978) and, less directly, Wason and Johnson-Laird (1972) investigate patterns of human nondeductive reasoning. Which of these may now succumb to formal analysis?

F. LOGIC PROGRAMMING

LOGIC PROGRAMMING refers to a family of higher level languages and an associated programming style based on writing programs as sets of assertions. These assertions are viewed as having *declarative* meaning as descriptive statements about entities and relations. In addition, the assertions derive a *procedural* meaning by virtue of being executable by an interpreter. Indeed, executing a logic program is much like performing a deduction on a set of facts.

A logic program consists of a set of clauses, where the general form of a clause is:

$$\langle \text{consequent} \rangle \text{ :- } \langle \text{antecedent}_1 \rangle, \langle \text{antecedent}_2 \rangle, \ldots, \langle \text{antecedent}_n \rangle$$

and each item in a clause is a positive literal, that is, an atomic formula $P(\text{term}_1, \ldots, \text{term}_n)$ for some predicate P. Not all clauses have antecedents.

A simple logic program for reversing a list is given by the following set of clauses:

```
APPEND(NIL,X,X)
APPEND(CONS(X,Y),Z,CONS(X,U)) :- APPEND(Y,Z,U)
REVERSE(NIL,NIL)
REVERSE(CONS(X,Y),Z) :- REVERSE(Y,R), APPEND(R,CONS(X,NIL),Z) .
```

Two observations must be made about this program: First, the terms involving CONS are not evaluated as they would be in LISP; rather, they are treated as symbolic objects. Second, both APPEND and REVERSE take one more argument than the corresponding LISP function. This is because APPEND(X,Y,Z) does not name a function but, rather, names the relation *Z is the result of appending X and Y*. Similarly, REVERSE(X,Y) means *Y is the result of reversing X*. One consequence of this is that a logic program, unlike its LISP counterpart, can often be run backwards. For example, the APPEND program could be used to find pairs of lists that, when concatenated, yield a given list.

To execute a logic program, we supply a *goal*, for example, REVERSE (CONS(A,CONS(B,CONS(C,NIL))),X). The interpreter finds substitutions for X that make the formula a consequence of the clauses in the program. This is done by cycling through the clauses, matching the goal against the consequent (by unification; see Article XII.B), recursively setting up antecedents as subgoals, and backtracking in case of failure. If all the subgoals can be satisfied, the goal is proved, and the substitutions found during matching constitute an answer. Forced backtracking can be used to produce systematically all substitutions that make the goal provable. For the goal above, the interpreter would find the substitution CONS(C,CONS(B,CONS(A,NIL))) for X.

One feature that distinguishes logic programming from ordinary theorem-proving is that, while the declarative semantics allow the clauses—and the antecedents within a clause—to appear in any order, the procedural interpretation is sensitive to the order. Thus, the programmer can rely on assertions being searched in sequence, top to bottom and left to right, and can structure a program for maximum efficiency.

Another difference between logic programming and general theorem-proving has to do with the restrictions on the form of the assertions themselves. In theorem-proving terminology, logic programs consist of sets of Horn clauses—disjunctive formulas with at most one positive literal. It is easy to see that the clauses of a logic program are Horn clauses: Any disjunction of the form $\neg A \vee \neg B \vee \cdots \vee \neg C \vee D$ can be rewritten as an equivalent implicational formula, $A \,\&\, B \,\&\, \cdots \,\&\, C \rightarrow D$, which is a notational variant of the form of clauses in logic programs.

By enforcing this restriction to Horn clauses, logic programming ensures relative tractability of deductions. It should be noted that, as with most very-high-level programming languages, it is not hard to write extremely inefficient logic programs—especially since the interpreter's basic strategy is exhaustive backtracking. Many implementations give the programmer some control over backtracking and allow the insertion of a special symbol (typically a slash, "/") between antecedents in a clause to prevent backtracking past that literal. This often improves efficiency, but at the expense of semantic purity, since some deductive consequences of the clauses may be underivable while other formulas, not logical consequences of the clauses, may be "deduced" from failure to derive a fact. (This latter case corresponds to the THNOT construct in the PLANNER languages.)

Logic Programming and AI

Although logic programming has been applied to diverse problems, some of which can hardly be considered exclusively AI problems (e.g., database management), there are at least two reasons why logic programming has special importance for AI. First, logic programming offers an alternative to LISP as a powerful language for symbol manipulation, apart from the semantic content of the symbols qua representations. The interpreters that drive logic programs do unification (Robinson, 1965b, and Article XII.B) and, thus, already incorporate the pattern-matching machinery that many applications require and that is programmed explicitly in LISP.

The second, and more important, reason why logic programming is of interest to AI has to do with its usefulness for knowledge representation. Predicate logic is a formalism considered by many to be a natural and powerful representation language marred only by its perceived computational inefficiency (see Article III.C1, in Vol. I). Any approach based on logic that can

demonstrate efficient execution (which logic programming does, in fact, claim) would be a serious candidate as a representation language.

To see how a logic program could be used to represent real-world knowledge, consider the following simple set of clauses:

```
SEES(X,Y) :- PERSON(X), PHYSOBJ(Y), OPEN(EYES(X)), IN-FRONT-OF(X,Y)
SEES(X,Y) :- PERSON(X), EVENT(Y), WATCHING(X, FILM-OF(Y))
PERSON(MOTHER(John))
EVENT(BIRTHDAY(Henry))
EVENT(GRADUATION(John))
WATCHING(MOTHER(John), FILM-OF(GRADUATION(John))) .
```

Consider the following three goals:

1. `SEES(MOTHER(John), GRADUATION(John))`
2. `SEES(MOTHER(U), GRADUATION(U))`
3. `SEES(U,V)`

These goals can be viewed as queries to a deductive question-answering system. The first can be paraphrased *Did John's mother see his graduation?*—a yes/no question. The second and third goals resemble "Wh-questions"—the free variables U and V indicating that the answer is to be the individual or individuals satisfying the condition. In particular, the second goal corresponds to the question *Who is it whose mother saw his graduation?* The third asks simply, *Who saw what?*

The logic-program interpreter would cycle through the asserted facts, matching the goal against the consequent and solving the antecedents as subgoals. If the subgoals can be satisfied, the goal is proved and the answer to the yes/no question will be YES. If, after exhaustively trying alternative facts, the goal still cannot be proved, the answer is NO. For goals with variables, the system can produce all substitutions that make the goal provable. With the clauses given above, the answer for goal 1 would be YES; the answer for goal 2 would be U = `John`; and the answer for goal 3 would be U = `MOTHER(John)`, V = `GRADUATION(John)`.

Development of Logic Programming and Current Status

The parallels between computation and logical proof have long been recognized, especially in the theory of computation. An interesting discussion of the many connections between logic and computation can be found in an early work of McCarthy (1963). In a sense, executing an applicative program, for example, a program in "pure" LISP, can be thought of as calculating the proof of an identity "$f(\text{arg}_1, \text{arg}_2, \ldots) = \text{result}$" by applying various axioms of identity according to a fixed control regime, much as the assertions of a logic program are applied.

Ordinarily, logic programming is understood to refer more narrowly to the style of programming introduced and advocated by Kowalski (1974, 1979), which was eventually incorporated into PROLOG, the best-known of the logic programming languages. PROLOG has several dialects and is supported in numerous installations in the United States, in Britain, and on the Continent. Especially active groups are in Edinburgh, London, Marseilles, and Budapest. Diverse applications have been programmed in PROLOG, including natural-language processing (Colmerauer et al., 1973), database retrieval (Warren, 1981), and program synthesis and planning (Warren, 1974).

PROLOG, and logic programming in general, has increased in popularity in recent years. In Europe, especially, PROLOG is a serious contender as the major AI implementation language. Much effort has been devoted to developing PROLOG compilers that compete favorably with LISP in efficiency of generated code (Warren, Pereira, and Pereira, 1977). In the United States, also, there has been interest in PROLOG, as well as in LOGLISP, a LISP-based logic-programming system developed at Syracuse University (Robinson and Sibert, 1980).

Conclusion

To a certain extent, the development of logic programming has followed the pattern of LISP. Both languages are founded on clear, mathematically motivated formalisms. Both languages have a side-effect-free kernel and a procedural interpretation that can be defined in a simple and elegant fashion. Yet both language families have yielded to the practical needs of their user communities and have incorporated numerous features that detract from their underlying elegance in favor of improved convenience and efficiency. In a sense, the fact that logic programming has progressed to the point of incorporating such features attests to its practicality and growing popularity.

References

Kowalski (1974, 1979) discusses logic programming and Warren, Pereira, and Pereira (1977) discuss the PROLOG language.

BIBLIOGRAPHY

Ballantyne, A. M., and Bennett, W. 1973. Graphing methods for topological proofs. Memo ATP-7, Mathematics Dept., University of Texas, Austin.

Ballantyne, A. M., and Bledsoe, W. W. 1975. Automatic proofs of theorems in analysis using non-standard techniques. Memo ATP-23, Mathematics Dept., University of Texas, Austin. (Also in *J. ACM*, July 1977.)

Ballantyne, A. M., and Lankford, D. 1979. New decision algorithms for finitely presented commutative semigroups. Memo MTP-4, Mathematics Dept., Louisiana Tech University.

Bibel, W., and Schreiber, J. 1974. Proof search in a Gentzen-like system of first order logic. Bericht Nr. 7412, Technische Universitat, Munich.

Black, F. 1968. A deductive question-answering system. In M. Minsky (Ed.), *Semantic information processing*. Cambridge, Mass.: MIT Press, 354–402.

Bledsoe, W. W. 1971. Splitting and reduction heuristics in automatic theorem proving. *Artificial Intelligence* 2:55–77.

Bledsoe, W. W. 1974. The sup-inf method in Presburger arithmetic. Memo ATP-18, Mathematics Dept., University of Texas, Austin.

Bledsoe, W. W. 1977. Non-resolution theorem proving. *Artificial Intelligence* 9:1–35.

Bledsoe, W. W., and Ballantyne, A. M. 1979. On automatic generation of counterexamples, Memo ATP-44A, Mathematics Dept., University of Texas, Austin.

Bledsoe, W. W., Bruell, P., and Shostak, R. 1979. A prover for general inequalities. Memo TPP-40A, Mathematics Dept., University of Texas, Austin.

Bledsoe, W. W., and Tyson, M. 1975. The UT interactive theorem prover. Memo ATP-17, Mathematics Dept., University of Texas, Austin.

Bobrow, D. G. (Ed.). 1980. Special issue on non-monotonic logic. *Artificial Intelligence* 13(1,2).

Boyer, R. S. 1971. *Locking: A restriction of resolution*. Doctoral dissertation, University of Texas, Austin.

Boyer, R. S., and Moore, J. S. 1979. *A computational logic*. New York: Academic Press.

Burstall, R. 1969. Proving properties of programs by structural induction. *Computer Journal* 12(1):41–48.

Chang, C., and Lee, R. C. 1973. *Symbolic logic and mechanical theorem proving*. New York: Academic Press.

Collins, A. M. 1978. Fragments of a theory of human plausible reasoning. *TINLAP-2*, 194–201.

Colmerauer, A., Kanoui, H., Pasero, R., and Roussel, P. 1973. Un systeme de communication homme-machine en français. In *Rapport, Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille, Luminy, France*.

Dacey, R. 1978. A theory of conclusions. *Philosophy of Science* 45:563–574.

de Kleer, J., et al. 1979. Explicit control of reasoning. In P. H. Winston and R. H. Brown (Eds.), *Artificial intelligence: An MIT perspective* (Vol. 1). Cambridge, Mass.: MIT Press, 93–116.

Doyle, J. 1979. A truth maintenance system. *Artificial Intelligence* 12:231–272.

Doyle, J. 1980. A model for deliberation, action, and introspection. Tech. Rep. AI-TR-581, AI Laboratory, Massachusetts Institute of Technology. (Doctoral dissertation.)

Doyle, J., and London, P. 1980. A selected descriptor-induced bibliography to the literature on belief revision. *SIGART Newsletter* 71:7–23.

Ernst, G. W. 1971. The utility of independent subgoals in theorem proving. *Information and Control* 18:237–252.

Ernst, G. W. 1973. A definition-driven theorem prover. *IJCAI 3.*

Gelernter, H. 1959. Realization of a geometry theorem-proving machine. *Proceedings of an International Conference on Information Processing.* Paris: UNESCO House, 273–282.

Gelernter, H. 1963. Realization of a geometry theorem proving machine. In E. A. Feigenbaum and J. Feldman (Eds.), *Computers and thought.* New York: McGraw-Hill, 134–152.

Green, C. 1969. Theorem-proving by resolution as a basis for question-answering systems. In B. Meltzer and D. Michie (Eds.), *Machine intelligence 4.* New York: American Elsevier, 183–205.

Gumb, R. D. 1978. Summary of research on computational aspects of evolving theories. *SIGART Newsletter* 67:13.

Gumb, R. D. 1979. *Evolving theories.* New York: Haven.

Hayes, P. J. 1973. Computation and deduction. *Symposium on the Mathematical Foundations of Computer Science, Czechslovakia Academy of Science,* 105–118.

Hewitt, C. 1971. Description and theoretical analysis (using schemata) of PLANNER: A language for proving theorems and manipulating models in a robot. Rep. No. AI-TR-258, AI Laboratory, Massachusetts Institute of Technology. (Doctoral dissertation.)

Hewitt, C. 1975. How to use what you know. *IJCAI 4,* 189–198.

Hewitt, C., et al. 1973. A universal modular actor formalism for artificial intelligence. *IJCAI 3,* 235–245.

Hintikka, J. 1971. Semantics for propositional attitudes. In L. Linsky (Ed.), *Reference and modality.* London: Oxford University Press, 145–167.

Huet, G. 1972. Constrained resolution: A complete method for higher order logic. Rep. No. 1117, Jennings Computing Center, Case Western Reserve University. (Doctoral dissertation.)

Huet, G. 1975. A unification algorithm for typed lambda calculus. *Theoretical Computer Science* 1:27–57.

Kling, R. 1971. A paradigm for reasoning by analogy. *Artificial Intelligence* 2:147–178.

Knuth, D. E., and Bendix, P. 1970. Simple word problems in universal algebras. In J. Leech (Ed.), *Computational problems in abstract algebra.* Oxford, England: Pergamon Press.

Kowalski, R. 1974. Predicate logic as a programming language. In J. L. Rosenfeld (Ed.), *Information processing 74*. Amsterdam: North-Holland, 569–574.

Kowalski, R. 1979. *Logic for problem solving*. New York: American Elsevier.

Kowalski, R., and Kuchner, D. 1971. Linear resolution with selector function. *Artificial Intelligence* 2:227–260.

Kripke, S. A. 1971. Semantical considerations on modal logic. In L. Linsky (Ed.), *Reference and modality*. London: Oxford University Press, 63–72.

Lankford, D. S. 1975. Complete sets of reductions for computational logic. Memo ATP-25, Mathematics Dept., University of Texas, Austin.

Lankford, D. S., and Ballantyne, A. M. 1977. Decision procedures for simple equational theories with commutative axioms: Complete sets of commutative reductions. Memo ATP-35, Mathematics Dept., University of Texas, Austin.

Lenat, D. B. 1976. AM: An artificial intelligence approach to discovery in mathematics as heuristic search. Rep. No. STAN-C[illegible]-76-570. Computer Science Dept., Stanford University. (Doctoral dissertation. Reprinted in R. Davis and D. B. Lenat. 1980. *Knowledge-based systems in artificial intelligence*. New York: McGraw-Hill.)

Loveland, D. 1978. *Automatic theorem proving: A logical basis*. Amsterdam: North-Holland.

Loveland, D., and Stickel, M. 1973. A hole in goal trees: Some guidance from resolution theory. *IJCAI 3*, 153–161.

McCarthy, J. 1963. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg (Eds.), *Computer programming and formal systems*. Amsterdam: North-Holland.

McCarthy, J. 1968. Programs with common sense. In M. Minsky (Ed.), *Semantic information processing*. Cambridge, Mass.: MIT Press, 403–409.

McCarthy, J. 1980. Circumscription—A form of non-monotonic reasoning. *Artificial Intelligence* 13:27–39.

McCorduck, P. 1979. *Machines who think*. San Francisco: Freeman.

McDermott, D. 1978. Planning and acting. *Cognitive Science* 2:71–109.

McDermott, D. 1980. Non-monotonic logic II: Non-monotonic modal theories. Rep. No. 174, Computer Science Dept., Yale University.

McDermott, D., and Doyle, J. 1980. Non-monotonic logic I. *Artificial Intelligence* 13:41–72.

Minsky, M. 1980. A framework for representing knowledge. In J. Haugeland (Ed.), *Mind design: Philosophy, psychology, and artificial intelligence*. Montgomery, Vt.: Bradford Books.

Moore, R. C. 1980a. *Reasoning from incomplete knowledge in a procedural deduction system*. New York: Garland.

Moore, R. C. 1980b. Reasoning about knowledge and action. Tech. Note 191, AI Center, SRI International, Inc., Menlo Park, Calif.

Nelson, C. G., and Oppen, D. 1978. Efficient decision procedures based on congruence closure. Memo AIM-309 (CS-646), Computer Science Dept., Stanford University.

Nevins, A. J. 1974. A human oriented logic for automatic theorem proving. *J. ACM* 21:606–621.

Nevins, A. J. 1975. Plane geometry theorem proving using forward chaining. *Artificial Intelligence* 6:1–23.

Newell, A., and Simon, H. A. 1956. The logic theory machine. *IRE Transactions on Information Theory* 2:61–79.

Nilsson, N. J. 1971. *Problem solving methods in artificial intelligence.* New York: McGraw-Hill.

Nilsson, N. J. 1980. *Principles of artificial intelligence.* Palo Alto, Calif.: Tioga.

Peterson, G. E., and Stickel, M. 1977. Complete sets of reductions for equational theories with complete unification algorithms. Memo, Computer Sciences Dept., University of Arizona.

Presburger, M. 1930. Uber die Vollstandigkeit eins gewissen Systems der Arithmetik ganser Zahlen in welchem die Addition als einsige Operation hervortritt. *Sprawozdanie z i kongresu matematykow krajow slowianskich, Warszawa (Comptes-rendus du I congrès des mathématiciens des pays slaves),* 92–101.

Quine, W. V., and Ullian, J. S. 1978. *The web of belief* (2nd ed.). New York: Random House.

Reiter, R. 1976. A semantically guided deductive system for automatic theorem proving. *IEEE Transactions on Computers* C-25.

Reiter, R. 1980. A logic for default reasoning. *Artificial Intelligence* 13:81–132.

Risch, R. 1969. The problem of integration in finite terms. *Transactions of the AMS* 139:167–189.

Robinson, G. A., and Wos, L. 1969. Paramodulation and theorem-proving in first order theories with equality. In D. Michie (Ed.), *Machine intelligence 4.* Edinburgh: Edinburgh University Press.

Robinson, J. A. 1965a. Automatic deduction with hyper-resolution. *International J. Computational Mathematics* 1:227–234.

Robinson, J. A. 1965b. A machine-oriented logic based on the resolution principle. *J. ACM* 12:23–41.

Robinson, J. A., and Sibert, E. E. 1980. Logic programming in LISP. School of Computer and Information Science, Syracuse University.

Shostak, R. S. 1975. On the completeness of the sup-inf method. SRI International, Inc., Menlo Park, Calif.

Skolem, T. 1967. The foundations of elementary arithmetic established by means of the recursive mode of thought, without the use of apparent variables ranging over infinite domains. In *From Frege to Goedel.* Cambridge, Mass.: Harvard University Press.

Tukey, J. W. 1960. Conclusions vs. decisions. *Technometrics* 2:423–433.

Wang, H. 1960. Toward mechanical mathematics. *IBM J. Research and Development* 4:2–22.

Warren, D. H. D. 1974. WARPLAN: A system for generating plans. Memo 76, Computational Logic Dept., School of Artificial Intelligence, University of Edinburgh.

Warren, D. H. D. 1981. Efficient processing of interactive relational database queries expressed in logic. *Proceedings of the Conference on Very Large Databases, Cannes, France,* 272–281.

Warren, D. H. D., Pereira, L. M., and Pereira, F. 1977. PROLOG—The language and its implementation compared with LISP. *Proceedings of the Symposium on Artificial Intelligence and Programming Languages (ACM); SIGPLAN Notices* 12(8); and *SIGART Newsletter* 64:109–115.

Wason, P. C., and Johnson-Laird, P. N. 1972. *Psychology of reasoning: Structure and content.* Cambridge, Mass.: Harvard University Press.

Weyhrauch, R. W. 1980. Prolegomena to a theory of mechanised formal reasoning. *Artificial Intelligence* 13:133–170.

Wos, L., Robinson, G. A., and Carson, D. F. 1965. Efficiency and completeness of the set of support strategy in theorem proving. *J. ACM* 12:536–541.

NAME INDEX

Pages on which an author's work is discussed are italicized.

SUBJECT INDEX