THEOREM PROVING

HISTORY, STATUS, AND FUTURE PROSPECTS

Automated theorem proving is the study of techniques for programming computers to search for proofs of formal assertions, either fully automatically or with varying degrees of human guidance. This discipline is potentially of tremendous value, because reasoning and inference underlie so many human activities. Automated (or mechanical) theorem proving is based on a foundation of formal logic that has been developed over the past several centuries by mathematicians and philosophers. This heritage of formal logic is often taken for granted, despite the tremendous amount of thought and effort that went into its development. The possibility that theorem proving programs can exist is closely tied to some properties of formal logic, which we now describe.

Formal Logic

The idea of formal logic is that the validity or correctness of an argument does not depend on the particular symbols used, but only on the form of the argument. Therefore, the test for whether an argument is correct is purely syntactic, consisting of checking whether the form of the argument is syntactically correct. Formal logic thus gives a systematic method for checking whether an argument is correct or not.

For example, consider the following argument:

All kachunks are groofy
Ork is a kachunk
Therefore Ork is groofy

We can know that the preceding inference is valid even without knowing what kachunks, Ork, and groofy mean. Any inference that has the form of the preceding one is correct, and this form can be checked by simple, syntactic methods. It is only necessary to verify that the inference follows a particular pattern. There are other correct forms of inference as well, and each one can be checked syntactically. A *proof* is an argument, or demonstration, that consists of a number of correct inferences put together.

The fact that formal logic has such simple syntactic tests for correctness of logical arguments gives more confidence in inferences that have been made and checked, since the procedure for checking is simple and unambiguous. This also means that it is possible to write a (simple) program to check if a proof is correct, and this program needs to know nothing about the meaning of the symbols or the intended area of application. The same proof checker can be used over and over again for different areas of application and different symbols. In addition, such proof checkers are themselves very simple in structure, increasing one's confidence in their reliability.

Checking that arguments are correct is of potential use for program and hardware verification, for example, because programs and computers are often used in situations where failure can be costly or even disastrous. Of course, a verification that a proof is correct does not necessarily imply that the program or hardware will work correctly, but it does help.

Finding Proofs

Mechanical theorem proving is concerned not only with checking the correctness of previously supplied proofs, but also with constructing proofs of valid statements in some formal logical system. Thus, given the task of proving that Ork is groofy, and given the axioms All kachunks are groofy and Ork is a kachunk, a theorem prover might derive the proof given previously. Of course, interesting theorems will have proofs that are much more complicated than this, consisting of many steps of argument concatenated together. Theorem provers typically construct proofs by searching through a large space of possibilities. The term *mechanical* means that the methods can be programmed on a computer.

Given a set A of axioms and a valid statement B, a theorem proving program should, ideally, eventually construct a proof of B from A. Given an invalid statement B, the program may run forever without coming to any definite conclusion. This is the best one can hope for, in general, and indeed even this is not possible always. In principle, theorem proving programs can be written just by enumerating all possible proofs and stopping when a proof of the desired statement is found, but this approach is very inefficient. Much more powerful methods have been developed.

There are many formal logical systems in which one can do theorem proving, all of which reduce the check for correctness of an inference to a simple syntactic check. To do theorem proving, it is necessary to choose some such formal logical system. One common system is Zermelo–Fraenkel set theory, which is the foundation of most of modern mathematics. Another common system is first-order logic, which we shall discuss later.

History of Mathematics

Mathematics was not always done as formally as it is today. In fact, the desire to mechanize proofs was one of the motives for formalizing mathematics. Thus theorem provers are a logical continuation of the process of formalizing mathematics and reasoning in general. Indeed, theorem provers are finding use in instruction in logic and mathematics, and this will probably increase to the point that many student assignments will consist of constructing proofs on the computer, assisted by theorem proving programs.

The initial, informal approach to mathematics began to change, partially as a result of the discovery of some logical paradoxes in mathematics, and partially because of difficulties in knowing when a mathematical argument was correct. Hilbert had as a goal the complete formalization of mathematics, so that one could know whether any mathematical statement was true or not by purely mechanical means. Though Hilbert's program of formalizing mathematics did not totally succeed, it did result in a further development of mathematical logic and in discoveries that would eventually lead to mechanical theorem proving.

One reason for the failure of Hilbert's program was the incompleteness theorem of Gödel, which states that in any sufficiently powerful logical system, there are statements that are true but not provable. This is a profound result and shows that it will never be possible to prove all of the true statements of mathematics. However, it is remarkable that many of the interesting statements of mathematics are provable. Furthermore, for some logics, such as first-order logic, there is a completeness result: All true statements *are* provable. This is because first-order logic is not powerful enough to express arithmetic. In addition, a partial fulfilment of Hilbert's program is still conceivable, using powerful theorem provers to verify many of the valid assertions.

Gödel's incompleteness theorem has an analogue in Turing machines and undecidability. Undecidability results concerning Turing machines can be used to provide a simple proof that there can be no sound (correct) theorem prover capable of finding proofs of all true statements of the form "Turing machine X will not halt on input Y." This is therefore an analogue of Gödel's theorem, but in a concrete form. The unprovable sentence that Gödel constructed is complicated, but a sentence about nonhalting of a Turing machine somehow has more intuitive appeal. Furthermore, the proof that such statements about Turing machines are unprovable is fairly simple and direct.

Despite such negative results, there is much that theorem provers can accomplish. There seems to be no a priori reason why they could not become as effective or more effective at this task than humans.

A Logic with Quantifiers

One of the most common logical systems in use, especially for mechanical theorem provers, is first-order logic, discovered by Frege, and we will emphasize it, too. First-order logic is in itself quite an accomplishment and, like formal mathematics in general, the result of a long process of development. It has a particularly simple syntax and semantics, but extreme flexibility and expressiveness. It would be ideal in many respects as a programming language if it could execute efficiently. This is in part the purpose of theorem provers: to provide an efficient execution mechanism for first-order logic and other logics.

In first-order logic, the logical connectives are $\wedge$, signifying conjunction (and); $\vee$, signifying disjunction (or); $\neg$, signifying negation (not); and $\supset$, signifying logical implication. There are also variables $x, y, z, \ldots$, constant symbols $a, b, c, \ldots$, function symbols $f, g, h, \ldots$, and predicate symbols $P, Q, R, \ldots$. A predicate is a property of objects that can be either true or false. These can be combined into logical formulas, so that, for example, the formula

$$P(x) \supset Q(f(x)) \vee R(g(x))$$

means "If P is true of x, then either Q is true of f applied to x, or R is true of g applied to x." There are also quantifiers; $\forall xA$ means, intuitively, "For all x, A is true" and $\exists xA$ means, intuitively, "There exists an x such that A is true." An example of a formula involving quantifiers is

$$\forall x(P(x) \supset \exists yQ(x, y))$$

In first-order logic, there is a formal definition of what it means for such a formula to be valid. There are also collections of inference rules that can be used to prove any valid formula. However, the question of whether a formula is valid is undecidable. But it is *partially* decidable in that it is possible to find a proof of any valid formula, given enough time. Thus it is possible to write a fairly simple computer program that will eventually find a proof of any valid formula of first-order logic. Despite this, first-order logic is surprisingly expressive.

Mechanizing Proof

Of course, humans were proving theorems long before the advent of computers, and still are, often more successfully. Why, then, the interest in theorem proving on computers?

One advantage of theorem proving on computers is the speed and accuracy of computers. Humans make mistakes and get tired. Also, there are applications like program verification where the theorems to be proven are not necessarily difficult, just boring and full of syntactic complexity. Such an area would seem to be an ideal application for computers.

But mechanizing theorem proving has turned out to be much harder than expected. There is still a mystery in how humans prove theorems so well that we have not yet begun to understand. This is probably because most current theorem proving methods are too syntactic—that is, they do not use information about what the symbols mean, and they do not learn from previous proof attempts. Nor do they have higher-level control over proof plans and strategies to guide their search attempt. So we have the curious situation that the very features that make logic attractive for mechanization may also explain its failure to provide powerful theorem provers. Logic enables us to throw away semantics when checking for correctness of proofs, but it could be that this semantic information is just what we need in order to make the search for a proof more efficient. Logic enables us to forget about the higher-level structure of proofs and concentrate instead on low-level inferences, but it could be that a broader view is what we need to make the proof search feasible on hard problems. There is some work in progress to build theorem provers that incorporate these more "human" features of reasoning.

History of Theorem Proving

Despite the potential advantages of machine theorem proving, it was difficult initially to obtain any kind of respectable performance from machines on theorem proving problems. We briefly survey some of the history of the development of automated theorem provers.

Some of the earliest theorem provers (1) as well as that of Prawitz were based on Herbrand's theorem, which gives an enumeration process for testing if a theorem of first-order logic is true. However, this approach turned out to be too inefficient. The *resolution* approach of Robinson (2,3) was developed in about 1963 and led to a significant advance in first-order theorem provers. This approach involved a *unification* algorithm, which essentially guided the enumeration process to find the formulas most likely to lead to a proof. Resolution is a machine-oriented inference step that is often difficult for humans to follow. The resolution inference rule in itself is all that is needed to program a theorem prover that can, in principle, prove all true theorems of first-order logic. In fact, all of the elements of resolution had been known for decades, but Robinson brought them to the fore at the right time. Wos et al. at Argonne National Laboratory took the lead in implementing resolution theorem provers, with some initial success on group theory problems that had been intractable before. They were even able to solve some previously open problems using resolution theorem provers.

The initial successes of resolution led to a rush of enthusiasm, as resolution theorem provers were applied to question-answering problems, situation calculus problems, and many others. It was soon discovered that the method had serious inefficiencies, and a long series of refinements were developed to attempt to overcome them. This included the unit preference rule, the set of support strategy, hyper-resolution, paramodulation for equality, and a nearly innumerable list of other refinements. Data structures were developed permitting the resolution operation to be implemented much more efficiently. There were also other strategies, such as model elimination (4), which led eventually to logic programming and Prolog. Some other attempts dealt with higher-order logic, such as the matings prover of Andrews (5) and the Boyer-Moore prover (6) for proofs by mathematical induction.

However, the initial enthusiasm for resolution, and for automated deduction in general, soon wore off. The initial feeling that all the problems of artificial intelligence could be solved by giving them to a resolution theorem prover soon led to a reaction in which the limitations of formal methods were stressed and an emphasis on procedural methods and specialized problem solvers predominated. This reaction led, for example, to the development of expert systems. Today, there seems to be a more balanced view, and formal methods and theorem proving seem to be accepted as part of the standard artificial intelligence (AI) tool kit.

Current Theorem Provers

Despite early difficulties, the power of theorem provers has continued to increase. Notable in this respect is Otter (7), which is widely distributed and coded in C with efficient data structures. The increasing speed of hardware has also significantly aided theorem provers. But the most powerful automatic provers seem to be largely syntactic in method and not to emphasize human-oriented strategies. A recent impetus was given to theorem proving research by William McCune's solution of the Robbins problem (8) by a resolution theorem prover derived from Otter. The Robbins problem is a first-order theorem involving equality that had been known to mathematicians for decades but that no one was able to solve. McCune's prover was able to find a proof after about a week of computation.

In addition to developing first-order provers, there has been work on other logics. This work has generally found much more application in industry than first-order theorem provers have.

The simplest logic typically considered is *propositional logic,* in which there are only predicate symbols (that is, Boolean variables) and logical connectives. Despite its simplicity, propositional logic has surprisingly many applications, such as in hardware verification and constraint satisfaction problems. Propositional provers have even found recent applications in planning. The general validity (respectively, satisfiability) problem of propositional logic is nondeterministic polynomial (NP) hard, which means that it does not, in all likelihood, have an efficient general solution. Nevertheless, there are propositional provers that are surprisingly efficient and becoming increasingly moreso. The satisfiability problem for propositional logic has been investigated (9), and it has been found that there is a 0-1 boundary; on one side there are formulas that are easy to deal with because they are likely to be satisfiable, and on the other side are formulas that are easy because they likely have short proofs of unsatisfiability. The hardest formulas are those in the middle.

Binary decision diagrams (BDDs) (10) are a particular form of propositional formulas for which efficient provers exist. BDDs are used in hardware verification and have initiated a tremendous surge of interest by industry in formal verification techniques.

Another restricted logic for which efficient provers exist is that of temporal logic, the logic of time. This has applications to concurrency. The model checking approach of Clarke (11) and others has proven to be particularly efficient in this area and has stimulated considerable interest by industry.

Other logical systems for which provers have been developed are the theory of equational systems, for which term-rewriting techniques lead to remarkably efficient theorem provers; mathematical induction; geometry theorem proving; constraints; higher-order logic; and set theory.

In addition to proving theorems, finding counterexamples is of increasing importance. This permits one to detect when a theorem is not provable, and thus one need not waste time attempting to find a proof. This is, of course, an activity that human mathematicians often engage in. These counterexamples are typically finite structures. For the so-called *finitely controllable* theories, running a theorem prover and a counterexample finder together yields a decision procedure, which theoretically can have practical applications to such theories.

Among the current applications of theorem provers we can list hardware verification, program verification, and program generation. For a more detailed survey, see the excellent report by Loveland (12). Among potential applications of theorem provers are planning problems, the situation calculus, and problems involving knowledge and belief.

There are a number of provers in prominence today, including Otter (7), the Boyer–Moore prover (6), Andrew's matings prover (5), the HOL prover (13), Isabelle (14), Mizer (15), NuPRL (16), and PVS (17). Many of these require substantial human guidance to find proofs. Provers can be evaluated on a number of grounds. One is *completeness;* can they, in principle, provide a proof of every true theorem? Another evaluation criterion is their performance on specific examples; in this regard, the TPTP problem set (18) is of particular value. Finally, one can attempt to provide an analytic estimate of the efficiency of a theorem prover on classes of problems (19). This gives a measure that is, to a large extent, independent of particular problems or machines.

Future Research

Future research areas in theorem proving include the incorporation of more humanlike methods. Current provers are mostly machine based, and their methods are not like those a person would use. Set theory is a good example of the problem; human-oriented methods that simply replace a predicate by its definition often greatly outperform resolution-based provers on set theory problems. We need to incorporate this kind of reasoning into our theorem provers, too. In addition, humans often use semantics when proving theorems. When proving a theorem about groups, a human will be imagining various groups. When proving a theorem about geometry, a human will be imagining various geometric figures. But a typical theorem prover only deals in symbols, regardless of the area of application. In addition, we should attempt to develop strategies that are *goal sensitive* (that is, in which all inferences are in some sense closely related to the particular theorem we are trying to prove). This enables us to prune the irrelevant inferences and increase efficiency. We need provers that are propositionally efficient. Resolution, for example, is tremendously inefficient on propositional problems, which is curious because there are very efficient techniques in this domain. Other topics of interest include learning, analogy, and abstraction, which have tremendous potential for leading to more powerful provers.

Whether any of these techniques will lead to a truly powerful theorem prover in the foreseeable future is anyone's guess. Still, the potential payoff is so large, even revolutionary, that any effort in this direction is well justified. It could even be the basis of another industrial revolution, with computation based on inference instead of bit pushing. This would have the advantage that computers would be more comprehensible, more flexible, and probably more reliable. Even a lesser amount of progress could make a significant impact on system reliability, research in mathematics and other formal areas, instruction, and artificial intelligence.

We begin the next section by presenting the syntax and semantics of propositional logic and some propositional theorem proving procedures. Next we consider the syntax and semantics of first-order logic and briefly survey some of the many first-order theorem proving methods, with particular attention to resolution. We also consider techniques for first-order logic with equality. Finally, we briefly discuss some other logics, and corresponding theorem proving techniques. Unfortunately, due to space restrictions, much additional material had to be omitted. However, enough detail is given for the reader to program a reasonable theorem prover. In the following discussion, we indicate the set difference of A and B by $A - B$; that is, $\{x : x \in A, x \notin B\}$.

PROPOSITIONAL LOGIC

Propositional logic has no variables (except Boolean variables), no function symbols, and no quantifiers, but it is useful for a surprising number of tasks. In addition, even though the satisfiability problem for propositional logic is NP complete, there are decision procedures for it that often work very fast in practice. In fact, there are decision procedures that run in expected polynomial time on reasonable probability distributions of propositional formulas.

Syntax

A *proposition* is a statement that can be either true or false. An example is the statement "It is raining." We denote propositions by the letters P, Q, and R. A *Boolean connective* may be used to combine propositions into *propositional formulas.* The Boolean connectives include the binary infix connectives $\wedge$, signifying conjunction (and); $\vee$, signifying disjunction (or); $\supset$, signifying logical implication; and $\equiv$, signifying equivalence. Also, $\neg$, signifying negation (not), is a unary Boolean connective. Examples of propositional formulas are

$$P \wedge (Q \vee P)$$
$$(P \vee Q) \equiv (Q \vee P)$$
$$(\neg P) \vee P$$

For example, if P is the proposition "It is raining" and Q is the proposition "I get wet," then $P \supset Q$ means "If it is raining then I get wet." If the propositions $\{P_1, P_2, \ldots, P_n\}$ include all the propositions that appear in a formula A, then we say that A is a formula *over* the propositions $\{P_1, P_2, \ldots, P_n\}$. Parentheses are used to indicate bindings of connectives but may be omitted using the usual rules of precedence. The constants **true** and **false** are called *truth values.*

Semantics

An *interpretation* (*valuation*) over the propositions $\{P_1, P_2, \ldots, P_n\}$. is a function from the propositions $\{P_1, P_2, \ldots, P_n\}$

to truth values. Thus there are 2^n interpretations over $\{P_1, P_2, \ldots, P_n\}$. If I is an interpretation and P is a proposition, we write $I \vDash P$ (I *satisfies* P) if $I(P) =$ **true** and $I \nvDash P$ if $I(P) =$ **false.** Thus one of the interpretations I over $\{P, Q\}$ is defined by $I(P) =$ **true** and $I(Q) =$ **false.** In addition, there are three more interpretations over $\{P, Q\}$. For a Boolean formula A, we define its truth in I by the meanings of the Boolean connectives, as follows:

$$I \vDash \neg A \text{ iff } I \nvDash A$$
$$I \vDash A \wedge B \text{ iff } I \vDash A \text{ and } I \vDash B$$
$$I \vDash A \vee B \text{ iff } I \vDash A \text{ or } I \vDash B$$
$$I \vDash A \supset B \text{ iff } I \vDash \neg A \text{ or } I \vDash B$$
$$I \vDash A \equiv B \text{ iff } I \vDash A \supset B \text{ and } I \vDash B \supset A$$

We say that a formula A over $\{P_1, P_2, \ldots, P_n\}$ is *satisfiable* if there is an interpretation I over $\{P_1, P_2, \ldots, P_n\}$ such that $I \vDash A$. If a formula A is not satisfiable, it is called *unsatisfiable* or *contradictory.* We say that a formula A over $\{P_1, P_2, \ldots, P_n\}$ is *valid* if for all interpretations I over $\{P_1, P_2, \ldots, P_n\}$, $I \vDash A$. We say that a formula A over $\{P_1, P_2, \ldots, P_n\}$ is *invalid* otherwise. A formula B is a *logical consequence* of A if the formula $A \supset B$ is valid. Two formulas A and B are *equivalent* iff the formula $A \equiv B$ is valid.

For example, the formula $P \vee \neg P$ is valid, and $P \wedge \neg P$ is unsatisfiable. The formula $P \vee Q$ is satisfiable but not valid. The formulas $P \wedge Q$ and $Q \wedge P$ are equivalent.

There are a number of simple relationships between these concepts. For example, a formula A is valid iff $\neg A$ is unsatisfiable, and if A is valid and B is a logical consequence of A, then B is valid. If A is valid and A and B are equivalent, then B is valid, too.

PROPOSITIONAL PROOF PROCEDURES

The main problem for theorem proving purposes is, given a formula A, to determine whether it is valid. Since A is valid iff $\neg A$ is unsatisfiable, we can determine validity if we can determine satisfiability. Many theorem provers test satisfiability instead of validity.

The problem of determining whether a Boolean formula A is satisfiable is one of the NP-complete problems. This means that the fastest algorithms known require an amount of time that is asymptotically exponential in the size of A. Also, it is not likely that faster algorithms will be found, although no one can prove that they do not exist.

Despite this negative result, there is a wide variety of methods in use for testing if a formula is satisfiable. One of the simplest is *truth tables.* For a formula A over $\{P_1, P_2, \ldots, P_n\}$, this involves testing for each of the 2^n valuations I over $\{P_1, P_2, \ldots, P_n\}$ whether $I \vDash A$. In general, this will require time at least proportional to 2^n to show that A is valid, but it may detect satisfiability sooner.

Clause Form

Many of the other satisfiability checking algorithms depend on conversion of a formula A to *clause form.* This is defined as follows: An *atom* is a proposition. A *literal* is an atom or an atom preceded by a negation sign. The two literals P and $\neg P$ are said to be *complementary* to each other. A *clause* is a disjunction of literals. A formula is in clause form if it is a conjunction of clauses. Thus the formula

$$(P \vee \neg R) \wedge (\neg P \vee Q \vee R) \wedge (\neg Q \vee \neg R)$$

is in clause form. This is also known as *conjunctive normal form.* We represent clauses by sets of literals and clause form formulas by sets of clauses, so that the preceding formula would be represented by the following set of sets:

$$\{\{P, \neg R\}, \{\neg P, Q, R\}, \{\neg Q, \neg R\}\}$$

A *unit clause* is a clause that contains only one literal. The *empty clause* { } is understood to represent **false.**

It is straightforward to show that for every formula A there is an equivalent formula B in clause form. Furthermore, there are well-known algorithms for converting any formula A into such an equivalent formula B. These involve converting all connectives to $\wedge$, $\vee$, and $\neg$, pushing $\neg$ to the bottom, and bringing $\wedge$ to the top. Unfortunately, this process of conversion can take exponential time and can increase the length of the formula by an exponential amount.

A number of people have noticed that the exponential increase in size in converting to clause form can be avoided by adding extra propositions representing subformulas of the given formula. For example, suppose we are given the formula

$$(P_1 \wedge Q_1) \vee (P_2 \wedge Q_2) \vee (P_3 \wedge Q_3) \vee \cdots \vee (P_n \wedge Q_n)$$

A straightforward conversion to clause form creates 2^n clauses of length n, and a formula of length at least $n2^n$. However, we can add the new propositions R_i, which are defined as $P_i \wedge Q_i$; then we obtain the new formula

$$(R_1 \vee R_2 \vee \cdots \vee R_n) \wedge ((P_1 \wedge Q_1) \equiv R_1) \wedge \cdots \wedge ((P_n \wedge Q_n) \equiv R_n)$$

When this formula is converted to clause form, we obtain a much smaller set of clauses and avoid the exponential size increase. The same technique works for any Boolean formula. This transformation, however, is only satisfiability preserving, but this is enough for theorem proving purposes.

Semantic Trees

To obtain a decision procedure for propositional logic that is more efficient than truth tables, we introduce *semantic trees.* A semantic tree over the propositions $\{P_1, P_2, \ldots, P_n\}$ is a binary tree in which the edges (arcs) are labeled with P_i or $\neg P_i$, and in which the two arcs leaving any vertex are labeled with complementary predicate symbols. Also, we require that no path from the root to a leaf encounter the same predicate symbol more than once. An example of a semantic tree over $\{P_1, Q_1, R\}$ is given in Fig. 1. Each vertex in this semantic tree corresponds to a *partial interpretation* (that is, a mapping from a subset of the predicate symbols to truth values). For example, the vertex labeled V in Fig. 1 corresponds to the partial interpretation mapping P to **true** and Q to **false,** since P and $\neg Q$ appear on the path from the root to V. The truth value of R is not determined by this partial interpretation.

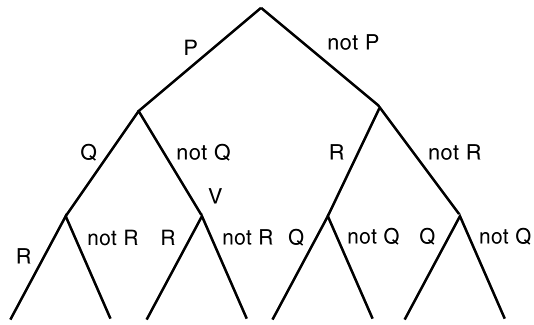

Figure 1. A semantic tree.

A Decision Procedure

We can use partial interpretations to make the satisfiability testing procedure more efficient. Instead of enumerating all the total interpretations, as in truth tables, it often suffices to enumerate only partial interpretations and to avoid giving more information than is necessary to determine the truth value of a formula A. Using this idea, we obtain a more efficient decision procedure. First we define A^I, where A is a Boolean formula and I is a partial interpretation, to be A with all occurrences of P replaced by **true,** for all propositions P such that I assigns **true** to P, and all occurrences of P replaced by **false,** for all propositions P such that I assigns **false** to P. Thus if I maps P to **true** and Q to **false,** and A is $P \vee Q \vee R$, then A^I is $\mathbf{true} \vee \mathbf{false} \vee R$. We also use the following *simplification rules* to eliminate occurrences of **true** and **false** from Boolean formulas, where X is an arbitrary Boolean formula:

$$
\begin{array}{ll}
X \wedge \mathbf{true} \rightarrow X & X \wedge \mathbf{false} \rightarrow \mathbf{false} \\
\mathbf{true} \wedge X \rightarrow X & \mathbf{false} \wedge X \rightarrow \mathbf{false} \\
X \vee \mathbf{true} \rightarrow \mathbf{true} & X \vee \mathbf{false} \rightarrow X \\
\mathbf{true} \vee X \rightarrow \mathbf{true} & \mathbf{false} \vee X \rightarrow X \\
\neg\mathbf{true} \rightarrow \mathbf{false} & \neg\mathbf{false} \rightarrow \mathbf{true}
\end{array}
$$

along with similar formulas for the other connectives $\equiv$ and $\supset$. We define $A^I \downarrow$ to be A^I with all of these simplifications applied. So if A^I is $\mathbf{true} \vee \mathbf{false} \vee R$, then $A^I \downarrow$ is **true.** Finally, we have the following decision procedure for testing if a (not necessarily clause form) Boolean formula A over set $\mathscr{P}$ of propositions is satisfiable:

procedure **Sat**(A,$\mathscr{P}$)
  [[test if Boolean formula A over $\mathscr{P}$ is satisfiable]]

  if A is **true** or **false then** return A;
  choose $P \in \mathscr{P}$ such that P appears in A;
  let I be the partial interpretation assigning **true** to P;
  let J be the partial interpretation assigning **false** to P;
  if Sat($A^I \downarrow$, $\mathscr{P} - \{P\}$) is **true then** return **true**;
  if Sat($A^J \downarrow$, $\mathscr{P} - \{P\}$) is **true then** return **true**;
  return **false**
end **Sat**;

This procedure essentially works by exploring the semantic tree and backtracking whenever the formula evaluates to **false.** We illustrate the working of this procedure on the formula

$$(P \wedge (P \supset Q)) \supset Q$$

Showing that this formula is valid is equivalent to showing that its negation is unsatisfiable. So we consider instead the formula A, which is

$$\neg[(P \wedge (P \supset Q)) \supset Q]$$

Suppose we choose the predicate symbol P first. Let I be the partial interpretation assigning **true** to P and let J be the partial interpretation assigning **false** to P. Then A^I is $\neg[(\mathbf{true} \wedge (\mathbf{true} \supset Q)) \supset Q]$. Simplifying, we obtain $\neg[(\mathbf{true} \supset Q) \supset Q]$ and then $\neg(Q \supset Q)$. This is because $(\mathbf{true} \supset Q)$ is equivalent to $(\neg\mathbf{true}) \vee Q$, which simplifies to Q. Thus $A^I \downarrow$ is $\neg(Q \supset Q)$. Similarly, A^J is $\neg[(\mathbf{false} \wedge (\mathbf{false} \supset Q)) \supset Q]$ and, simplifying, we obtain $A^J \downarrow$ as **false.** For $A^I \downarrow$, we consider the two formulas with Q replaced by **true** and **false,** respectively. Both of these simplify to **false.** Thus the formula A is unsatisfiable. This implies that the original formula $(P \wedge (P \supset Q)) \supset Q$ is valid.

The idea of exploring the semantic tree and backtracking is also essentially the idea of the Davis and Putnam procedure (20,21), which, however, is specialized to clause form and has a couple of additional rules. For example, if A is in clause form and one of the clauses in A is a unit clause (that is, contains only one literal), then P will be chosen to be a proposition that appears in a unit clause, and the order of considering I and J will be optimized for this case. There is also a *purity rule* for literals L in A such that the complement of L does not appear in A: If A is a set of clauses and C is a clause in A having a pure literal, then A is satisfiable iff $A - \{C\}$ is satisfiable. The Davis and Putnam procedure and its refinements, a number of which have rules for choosing P carefully, are often very efficient on propositional formulas. The reason is as follows: If there are many interpretations I such that $I \models A$, then **Sat** will probably find one quickly and thus will quickly detect that A is satisfiable. If, for fairly small partial interpretations I, we have that $A^I \downarrow$ is **false,** then **Sat** will not have to explore far and will quickly determine that A is unsatisfiable. The hardest formulas are those at the 0-1 boundary, where neither of these conditions is true. This aspect of satisfiability testing has been explored in (9).

Ground Resolution

Many first-order theorem provers are based on resolution, and there is a propositional analogue of resolution called *ground resolution,* which we now present. Although resolution is reasonably efficient for first-order logic, it turns out that ground resolution is generally much less efficient than Davis and Putnam-like procedures for propositional logic. However, we present ground resolution here as an introduction to first-order resolution.

Ground resolution is a decision procedure for propositional formulas in clause form. If C_1 and C_2 are two clauses and $L_1 \in C_1$ and $L_2 \in C_2$ are complementary literals, then

$$(C_1 - \{L_1\}) \cup (C_2 - \{L_2\})$$

is called a *resolvent* of C_1 and C_2. There may be more than one resolvent of two clauses, or maybe none. It is straightforward to show that a resolvent D of two clauses C_1 and C_2 is a logical consequence of $C_1 \wedge C_2$.

For example, if C_1 is $\{\neg P, Q\}$ and C_2 is $\{\neg Q, R\}$, then we can choose L_1 to be Q and L_2 to be $\neg Q$. Then the resolvent is $\{\neg P, R\}$. We also note that R is a resolvent of $\{Q\}$ and $\{\neg Q, R\}$, and $\{\,\}$ (the empty clause) is a resolvent of $\{Q\}$ and $\{\neg Q\}$.

A *resolution proof* of a clause C from a set S of clauses is a sequence $C_1, C_2, \ldots, C_n$ of clauses in which each C_i is either a member of S or a resolvent of C_j and C_k, for j, k less than i, and C_n is C. Such a proof is called a (resolution) *refutation* if C_n is $\{\,\}$. We have the following completeness result for resolution:

Theorem 3.1. Suppose S is a set of propositional clauses. Then S is unsatisfiable iff there exists a resolution refutation from S.

As an example, let S be the set of clauses

$$\{\{P\}, \{\neg P, Q\}, \{\neg Q\}\}$$

We then have the following resolution refutation from S, listing with each resolvent the two clauses that are resolved together:

1. P given
2. $\neg P, Q$ given
3. $\neg Q$ given
4. Q 1, 2, resolution
5. $\{\,\}$ 3, 4, resolution

(Here we omit set braces, except for the empty clause.) This is a resolution refutation from S, so S is unsatisfiable.

Define $\mathscr{R}(S)$ to be the set of clauses C such that there are clauses C_1 and C_2 in S such that C is a resolvent of C_1 and C_2. Define $\mathscr{R}^1(S)$ to be $\mathscr{R}(S)$ and $\mathscr{R}^{i+1}(S)$ to be $\mathscr{R}(S \cup \mathscr{R}^i(S))$, for $i > 1$. Typical resolution theorem provers essentially generate all of the resolution proofs from S (with some improvements that we will discuss later), looking for a proof of the empty clause. Formally, such provers generate $\mathscr{R}^1(S)$, $\mathscr{R}^2(S)$, $\mathscr{R}^3(S)$ and so on, until for some i, $\mathscr{R}^i(S), = \mathscr{R}^{i+1}(S)$, or the empty clause is generated. In the former case, S is satisfiable. If the empty clause is generated, S is unsatisfiable.

It is known that propositional (ground) resolution is much less efficient than Davis and Putnam's method as a decision procedure for satisfiability of formulas in the propositional calculus. Also, Haken (22) showed that there are unsatisfiable sets S of propositional clauses for which the length of the shortest resolution refutation is exponential in the size (number of clauses) in S. Despite these inefficiencies, we introduced propositional resolution as a way to lead up to first-order resolution, which has significant advantages.

FIRST-ORDER LOGIC

We now introduce the syntax and semantics of first-order logic. This is a much more powerful language than propositional logic, and theorem proving techniques for first-order logic are somewhat more complex, as well. Also, it is not always true that the same strategies that are most efficient for propositional logic are also most efficient for first-order logic.

Syntax

In first-order logic, there are *variables* (individual variables), denoted by the letters x, y, z, u, v, w; *function symbols* (function constants), denoted by the letters f, g, h; *predicate symbols* (predicate constants), denoted by the letters P, Q, R; and *Boolean connectives,* as for propositional logic. Each function and predicate symbol has an *arity,* which is an integer specifying how many arguments it takes. Function symbols of arity zero are often referred to as *constant symbols,* or *individual constants,* and denoted by the letters a, b, c, d. In addition, there are the *quantifiers* $\exists$ and $\forall$.

An example of a first-order formula is $\forall y \exists x P(x, y)$. If we interpret $P(x, y)$ as "x loves y," then this means "For all y, there exists an x such that x loves y." If we interpret $g(x)$ as "the mother of x" and $Q(y)$ as "y is female," then $\forall x Q(g(x))$ means "For all x, the mother of x is female." It should be clear that first-order logic is a highly expressive language. We now give a formal definition.

A *term* is defined inductively as a variable or a constant symbol or an expression of the form $f(t_1, \ldots, t_n)$, where the t_i are terms and f is a function symbol of arity n. An *atom* is an expression of the form $P(t_1, \ldots, t_n)$, where t_i are terms and P is a predicate symbol of arity n. The formulas of first-order logic are defined inductively as follows:

- If A is an atom, then A is a formula.
- If A is a formula, then $\neg A$ is a formula.
- If A and B are formulas, then $(A \wedge B)$, $(A \vee B)$, $(A \supset B)$, and $(A \equiv B)$ are also formulas.
- If A is a formula and x is a variable, then $(\forall x)A$ and $(\exists x)A$ are formulas.

Examples of formulas are $P(f(x), c)$, $P(x) \vee Q(y, f(x))$, and $(\forall x)(P(x) \supset (\exists y)Q\,(x, y))$. We note that parentheses are often omitted when not necessary for understanding. A *literal* is a formula of the form A or $\neg A$, where A is an atom. A formula without quantifiers is said to be *quantifier free.* In a formula of the form $(\forall x)A$, A is called the *scope* of the quantifier $(\forall x)$; similarly, in a formula of the form $(\exists x)A$, A is called the scope of the quantifier $(\exists x)$. A variable x in a formula A is *bound* if it is in the scope of some quantifier $\forall x$ or $\exists x$. If a variable is not bound, it is free. Thus in the formula $\forall x(P(x) \vee Q(y))$, the occurrence of x in $P(x)$ is bound, but the occurrence of y in $Q(y)$ is free. A formula without free variables is called a *sentence.* We sometimes write $(Q\ x)$ to refer to either $(\exists x)$ or $(\forall x)$. A quantifier free formula is in *conjunctive normal form* if it is a conjunction of disjunctions of literals (as in the propositional calculus); it is in *prenex conjunctive normal form* if it is of the form $(Q_1x_1)\,(Q_2x_2)\ldots(Q_nx_n)A$, where A is a quantifier free formula in conjunctive normal form. We sometimes call A the *matrix* of this formula.

Semantics

An *interpretation* (structure) I consists of a *domain* D, which is a nonempty collection (informally, a set) of objects, together with assignments of meanings to variables, constants, functions, and predicates. To a variable x, I assigns an element

x^I of D; to an individual constant a, I assigns an element a^I of D; to a function constant f, I assigns a function f^I from D^n to D, where n is the arity of f; and to a predicate constant P, I assigns a function P^I from D^n to {**true, false**}, where n is the arity of P. Given an interpretation I and a formula A, I assigns a truth value to A by interpreting Boolean connectives as in propositional logic and quantifiers consistent with their readings "for all" and "there exists." Formally, we define the meaning A^I of a term A in interpretation I as follows:

- If A is a variable or individual constant, then A^I is as stated previously.
- If $t_1, \ldots, t_n$ are terms and f is a function symbol of arity n, then $f(t_1 \ldots t_n)^I$ is $f^I(t_1^I \ldots t_n^I)$. Thus the meaning of f is a function that is applied to the meanings of the t_i. So A^I is an element of D, the domain of I.

Also, we define the truth value of a formula A in an interpretation I. We write $I \models A$, read "I satisfies A," to indicate that A is true in interpretation I. For interpretations I and J with domain D, and for variable x, we say that $I \equiv J \pmod{x}$ iff A^I and A^J are identical for all function symbols and predicate symbols A and for all variables A distinct from the variable x. We define $\models$ recursively as follows:

- If P is a predicate symbol of arity n and the t_i are terms, then $I \models P(t_1 \ldots t_n)$ if $P^I(t_1^I \ldots t_n^I)$ is **true.**
- $I \models (A_1 \vee A_2)$ iff $I \models A_1$ or $I \models A_2$.
- $I \models (A_1 \wedge A_2)$ iff $I \models A_1$ and $I \models A_2$.
- $I \models \neg A$ iff not $I \models A$.
- $I \models (A \supset B)$ iff $I \models ((\neg A) \vee B)$.
- $I \models (A \equiv B)$ iff $I \models ((A \supset B) \wedge (B \supset A)$.
- $I \models (\forall x)A$ iff for all interpretations J such that $J \equiv I \pmod{x}$, $J \models A$.
- $I \models (\exists x)A$ iff there exists an interpretation J such that $J \equiv I \pmod{x}$, $J \models A$.

For example, if the domain D of I is {0, 1, 2, . . .} and f^I is the successor function and P^I is the predicate testing if an integer is even, then $I \models (\forall x)(P(x) \vee P(f(x)))$ but not $I \models (\forall x)(P(x) \supset P(f(x)))$. We can see this because $P^I(n)$ is true if n is even, and $f^I(n)$ is $n + 1$. Thus for all n, either $P^I(n)$ is **true** or $P^I(f^I(n))$ is **true.** Thus for all J such that $J \equiv I \pmod{x}$, $J \models P(x)$ or $J \models P(f(x))$. Thus for all J such that $J \equiv I \pmod{x}$, $J \models (P(x) \vee P(f(x)))$. Thus $I \models (\forall x)(P(x) \vee P(f(x)))$. We write $I \not\models B$ to indicate that $I \models B$ does not hold; that is, I does not satisfy B.

We say A is satisfiable if there is an interpretation I such that $I \models A$; otherwise A is unsatisfiable, or a *contradiction.* If I satisfies A, we call I a *model* of A. We say A is *valid* if all interpretations I satisfy A. This is also written $\models A$. We say that B is a *logical consequence* (or a valid consequence) of A, written $A \models B$, if all interpretations that satisfy A also satisfy B. For example, $P(a) \models (\exists x)P(x)$. We write $A_1 A_2 \ldots A_n \models B$ to indicate that all models that satisfy all the A_i also satisfy B.

Sometimes the intended meaning of a first-order formula is not as obvious as one would like. For example, we can define a set S of axioms for addition and multiplication, as follows:

$$(\forall x)(x + 0 = x)$$
$$(\forall x)(\forall y)(x + s(y) = s(x + y))$$
$$(\forall x)(x * 0 = 0)$$
$$(\forall x)(\forall y)(x * s(y) = (x * y) + x)$$

Here we are using infix notation for $+$ and $*$ and using equality, which has not yet been defined. What is unusual is that every interpretation I satisfying S and such that $=$ is interpreted as the identity predicate on D will interpret $+$ as addition and $*$ as multiplication on the integers, where the integer i is expressed as the term $s^i(0)$. However, despite this, we cannot prove simple identities such as $x + y = y + x$; in fact, there are interpretations I satisfying S and not satisfying $\forall x \forall y(x + y = y + x)$. This curious fact is due to the existence of "nonstandard" elements of D that cannot be expressed as $s^i(0)$ for any integer i. To prove that addition is commutative, one needs to use mathematical induction. It can be very difficult for beginners to know when a formula is valid in first-order logic and when it requires mathematical induction to prove.

In fact, there is no general procedure for deciding first-order validity; it is known that validity in first-order logic is partially decidable but not decidable. This means that there is a procedure that, given any valid formula, will eventually halt and state that the formula is valid, but given an invalid formula, might not halt. However, it is known that there can be no recursive bound on the running time of such a procedure on valid formulas; thus the procedure may run a long time even on short formulas.

Some general results about models of first-order formulas are known. For example, if S is an infinite set of first-order formulas, none of which has free variables, and if no interpretation satisfies all formulas of S, then there is a finite subset $\{A_1, A_2, \ldots, A_n\}$ of S such that the formula $A_1 \wedge A_2 \wedge \cdots \wedge A_n$ is unsatisfiable. This is called *compactness.*

Another interesting fact about first-order models is that any first-order formula A that is satisfiable has a model I with domain D such that D is countable. Thus it is never necessary to consider uncountable domains in first-order logic, which implies that first-order logic cannot really express the existence of infinities beyond ω.

FIRST-ORDER PROOF SYSTEMS

We now discuss methods for partially deciding validity. These construct proofs of first-order formulas, and a formula is valid iff it can be proven in such a system. Thus there are complete proof systems for first-order logic, and Gödel's incompleteness theorem does not apply to first-order logic. Since the set of proofs is countable, we can partially decide validity of a formula A by enumerating the set of proofs and stopping whenever a proof of A is found. This already gives us a theorem prover, but provers constructed in this way are typically very inefficient.

There are a number of classical proof systems for first-order logic: Hilbert-style systems, Gentzen-style systems, natural deduction systems, semantic tableau systems, and others

(23). Since these generally have not found much application to automated deduction, except for semantic tableau systems, we do not emphasize them here. Typically they specify inference rules of the form

$$\frac{A_1, A_2, \ldots, A_n}{A}$$

which means that if we have already derived the formulas $A_1, A_2, \ldots, A_n$, then we can also infer A. Using such rules, we build up a proof as a sequence of formulas, and if a formula B appears in such a sequence, we have proved B.

We now discuss proof systems that have found application to automated deduction.

Clause Form

Many first-order theorem provers convert a first-order formula to clause form before attempting to prove it. The beauty of clause form is that it makes the syntax of first-order logic, already quite simple, even simpler. Quantifiers are omitted, and Boolean connectives as well. In the end we have just sets of sets of literals. It is amazing that the expressive power of first-order logic can be reduced to such a simple form. This simplicity also makes clause form suitable for machine implementation of theorem provers. Not only that, but the validity problem is also simplified in a theoretical sense; we only need to consider the Herbrand interpretations, so the question of validity becomes easier to analyze.

Any first-order formula A can be transformed to a clause form formula B such that A is satisfiable iff B is satisfiable. The translation is not validity preserving. So in order to show that A is valid, we translate $\neg A$ to clause form B and show that B is unsatisfiable. For convenience, we assume that A is a *sentence* (that is, it has no free variables).

The translation of a first-order sentence A to clause form has several steps:

- Push negations in.
- Replace existentially quantified variables by Skolem functions.
- Move universal quantifiers to the front.
- Convert the matrix of the formula to conjunctive normal form.
- Remove universal quantifiers and Boolean connectives.

We present this transformation in terms of sets of rewrite rules. A rewrite rule $X \to Y$ means that a subformula of the form X is replaced by a subformula of the form Y.

The following rewrite rules push negations in.

$$\begin{aligned}
(A \equiv B) &\to (A \supset B) \wedge (B \supset A)\\
(A \supset B) &\to ((\neg A) \vee B)\\
\neg\neg A &\to A\\
\neg(A \wedge B) &\to (\neg A) \vee (\neg B)\\
\neg(A \vee B) &\to (\neg A) \wedge (\neg B)\\
\neg(\forall x)A &\to (\exists x)(\neg A)\\
\neg(\exists x)A &\to (\forall x)(\neg A)
\end{aligned}$$

After negations have been pushed in, we assume for simplicity that variables in the formula are renamed so that each variable appears in only one quantifier. We then eliminate existential quantifiers by replacing formulas of the form $(\exists x)A[x]$ by $A[f(x_1, \ldots, x_n)]$, where $x_1, \ldots, x_n$ are all the universally quantified variables whose scope includes the formula A, and f is a new function symbol (that does not already appear in the formula).

The following rules then move quantifiers to the front:

$$\begin{aligned}
((\forall x)A) \vee B &\to (\forall x)(A \vee B)\\
B \vee (\forall x)A) &\to (\forall x)(B \vee A)\\
((\forall x)A) \wedge B &\to (\forall x)(A \wedge B)\\
B \wedge ((\forall x)A) &\to (\forall x)(B \wedge A)
\end{aligned}$$

Next, we convert the matrix to conjunctive normal form by the following rules:

$$\begin{aligned}
(A \vee (B \wedge C)) &\to (A \vee B) \wedge (A \vee C)\\
((B \wedge C) \vee A) &\to (B \vee A) \wedge (C \vee A)
\end{aligned}$$

Finally, we remove universal quantifiers from the front of the formula and replace a conjunctive normal form formula of the form

$$(A_1 \vee A_2 \vee \cdots \vee A_k) \wedge (B_1 \vee B_2 \vee \cdots \vee B_m) \wedge \cdots \wedge (C_1 \vee C_2 \vee \cdots \vee C_n)$$

by the set of sets of literals

$$\{\{A_1, A_2, \ldots, A_k\}, \{B_1, B_2, \ldots, B_m\}, \ldots, \{C_1, C_2, \ldots, C_n\}\}$$

This last formula is the clause form formula that is satisfiable iff the original formula is.

As an example, consider the formula

$$(\forall x)(\neg(P(x) \supset (\forall y)Q(x, y)))$$

First, we push negations in, which involves replacing $\supset$ by its definition as follows:

$$(\forall x)\neg((\neg P(x)) \vee (\forall y)Q(x, y))$$

Then we move $\neg$ in past $\vee$:

$$(\forall x)((\neg\neg P(x)) \wedge \neg(\forall y)Q(x, y))$$

Next we eliminate the double negation and move $\neg$ past the quantifier:

$$(\forall x)(P(x) \wedge (\exists y)\neg Q(x, y))$$

Now, negations have been pushed in. We note that no variable appears in more than one quantifier, so it is not necessary to rename variables. Next, we replace the existential quantifier by a Skolem function:

$$(\forall x)(P(x) \wedge \neg Q(x, f(x)))$$

There are no quantifiers to move to the front. Eliminating the universal quantifier, we obtain

$$P(x) \wedge \neg Q(x, f(x))$$

The clause form is then

$$\{\{P(x)\}, \{\neg Q(x, f(x))\}\}$$

We recall that if B is the clause form of A, then B is satisfiable iff A is. As in propositional calculus, the clause form translation can increase the size of a formula by an exponential amount. This can be avoided, as in the propositional calculus, by introducing new predicate symbols for subformulas. Suppose A is a formula with subformula B, so we write $A[B]$. Let $x_1, x_2, \ldots, x_n$ be the free variables in B. Let P be a new predicate symbol (that does not appear in A). Then we can transform $A[B]$ to the formula $A[P(x_1, x_2, \ldots, x_n)] \wedge (\forall x_1 \forall x_2 \ldots \forall x_n)(P(x_1, x_2, \ldots, x_n) \equiv B)$. Thus the occurrence of B in A is replaced by $P(x_1, x_2, \ldots, x_n)$, and the equivalence of B with $P(x_1, x_2, \ldots, x_n)$ is added on to the formula as well. This transformation can be applied to the new formula in turn, and again as many times as desired. The transformation is satisfiability preserving, which means that the resulting formula is satisfiable iff the original formula A was.

Free variables in a clause are assumed to be universally quantified. Thus the clause $\{\neg P(x), Q(f(x))\}$ represents the formula $\forall x(\neg P(x) \vee Q(f(x)))$. A term, literal, or clause not containing any variables is said to be *ground*.

A set of clauses represents the conjunction of the clauses in the set. Thus the set $\{\{\neg P(x), Q(f(x))\}, \{\neg Q(y), R(g(y))\}, \{P(a)\}, \{\neg R(z)\}\}$ represents the formula $(\forall x(\neg P(x) \vee Q(f(x)))) \wedge (\forall y(\neg Q(y) \vee R(g(y)))) \wedge P(a) \wedge (\forall z \neg R(z))$.

Herbrand Interpretations

There is a special kind of interpretation that turns out to be significant for mechanical theorem proving. This is called a *Herbrand interpretation.* Herbrand interpretations are defined relative to a set S of clauses. The domain D of a Herbrand interpretation I consists of the set of terms constructed from function and constant symbols of S, with an extra constant symbol added if S has no constant symbols. The constant and function symbols are interpreted so that for any finite term t composed of these symbols, t^I is the term t itself, which is an element of D. Thus if S has a unary function symbol f and a constant symbol c, then $D = \{c, f(c), f(f(c)), f(f(f(c))), \ldots\}$ and c is interpreted so that c^I is the element c of D and f is interpreted so that f^I applied to the term c yields the term $f(c)$, f^I applied to the term $f(c)$ of D yields $f(f(c))$, and so on. Thus these interpretations are quite syntactic in nature. There is no restriction, however, on how a Herbrand interpretation I may interpret the predicate symbols of S.

The interest in Herbrand interpretations for theorem proving comes from the following result:

Theorem 2. If S is a set of clauses, then S is satisfiable iff there is a Herbrand interpretation I such that $I \vDash S$.

What this theorem means is that for purposes of testing satisfiability of clause sets, one only needs to consider Herbrand interpretations. This implicitly leads to a mechanical theorem proving procedure, as we shall see. But first we need some terminology.

A *substitution* is a mapping from variables to terms that is the identity on all but finitely many variables. If L is a literal and α is a substitution, then $L\alpha$ is the result of replacing all variables in L by their image under α. We define the application of substitutions to terms, clauses, and sets of clauses similarly. We indicate by $\{x_1 \mapsto t_1, x_2 \mapsto t_2, \ldots, x_n \mapsto t_n\}$ the substitution mapping the variable x_i to the term t_i, for $1 \leq i \leq n$. For example, $P(x, f(x))\{x \mapsto g(y)\} = P(g(y), f(g(y)))$.

If L is a literal and α is a substitution, then $L\alpha$ is called an *instance* of L. Thus $P(g(y), f(g(y)))$ is an instance of $P(x, f(x))$. Similar terminology applies to clauses and terms.

If S is a set of clauses, then a *Herbrand set* for S is an unsatisfiable set T of ground clauses such that for every clause D in T there is a clause C in S such that D is an instance of C. If there is a Herbrand set for S, then S is unsatisfiable. For example, let S be the following clause set:

$$\{\{P(a)\}, \{\neg P(x), P(f(x))\}, \{\neg P(f(f(a)))\}\}$$

For this set of clauses, we have the following Herbrand set:

$$\{\{P(a)\}, \{\neg P(a), P(f(a)\}, \{\neg P(f(a)), P(f(f(a)))\}, \{\neg P(f(f(a)))\}\}$$

The *ground instantiation problem* is the following: Given a set S of clauses, is there a Herbrand set for S?

The following result is known as the Skolem-Herbrand-Gödel theorem, and it follows from Theorem 2:

Theorem 3. A set S of clauses is unsatisfiable iff there is a Herbrand set T for S.

It follows from this result that a set S of clauses is unsatisfiable iff the ground instantiation problem for S is solvable. Thus we have reduced the problem of first-order validity to the ground instantiation problem. This is actually quite an achievement, because the ground instantiation problem deals only with syntactic concepts, such as replacing variables by terms, and with propositional unsatisfiability, which is easily understood.

We also have the following theorem proving method as a result: Given a set S of clauses, let $C_1, C_2, C_3, \ldots$ be an enumeration of all of the ground instances of clauses in S. This set of ground instances is countable, so it can be enumerated. Consider the following procedure **Prover:**

```
procedure Prover(S)
  for i = 1, 2, 3, . . . do
    if {C1, C2, . . . Ci} is unsatisfiable then
      return ''unsatisfiable'' fi
  od
end Prover
```

By Herbrand's theorem, it follows that **Prover**(S) will eventually return "unsatisfiable" iff S is unsatisfiable. Thus we already have a primitive theorem proving procedure. It is interesting that some of the earliest attempts to mechanize theorem proving (1) were based on this idea. The problem with this approach is that it enumerates many ground instances that could never appear in a proof. However, the efficiency of propositional decision procedures is an attractive feature of this procedure, and it may be possible to modify it to obtain an efficient theorem proving procedure. In fact, many of the theorem provers in use today are based implicitly on this procedure, and thereby on Herbrand's theorem.

Unification and Resolution

Most mechanical theorem provers today are based on unification, which guides the instantiation of clauses in an attempt to make the procedure **Prover** more efficient. The idea of unification is to find those instances that are in some sense the most general ones that could appear in a proof. This avoids a lot of work that results from the generation of irrelevant instances by **Prover.**

In the following discussion we will use $\equiv$ to refer to syntactic identity of terms, literals, and so on. A substitution α is called a *unifier* of literals L and M if $L\alpha \equiv M\alpha$. If such a substitution exists, we say that L and M are *unifiable.* A substitution α is a most general unifier of L and M if for any other unifier β of L and M there is a substitution γ such that $L\beta \equiv L\alpha\gamma$ and $M\beta \equiv M\alpha\gamma$.

It turns out that if two literals L and M are unifiable, then there is a most general unifier of L and M, and such most general unifiers can be computed efficiently by a number of simple algorithms. The earliest in recent history was given by Robinson (3).

We present a simple unification algorithm on terms; this algorithm is similar to that presented by Robinson and is actually exponential time on large terms, but often efficient in practice. Algorithms that are most efficient (and even linear time) on large terms have been devised.

```
procedure Unify(r, s);
  [[return the most general unifier of terms r
    and s]]
  if r is a variable then
    if r ≡ s then return {} else
    (if r occurs in s then return fail else
      return {r ↦ s}) else
  if s is a variable then
    (if s occurs in r then return fail else
      return {s ↦ r}) else
  if the top-level function symbols of r and s
    differ or have different arities then return
      fail
  else
    suppose r is f(r1 . . . rn) and s is f(s1
      . . . sn);
    return(Unify_lists([r1 . . . rn], [s1 . . .
      sn]))
end Unify;

procedure Unify_lists([r1 . . . rn], [s1 . . .
  sn]);
  if [r1 . . . rn] is empty then return {}
    else
      θ ← Unify(r1, s1);
      if θ ≡ fail then return fail fi;
      α ← Unify_lists([r2 . . . rn]θ, [s2 . . .
        sn]θ)}
      if α ≡ fail then return fail fi;
  return {θ ∘ α}
end Unify_lists;
```

For this last procedure, we define $\theta \circ \alpha$ as the composition of the substitutions θ and α, defined by $t(\theta \circ \alpha) = (t\theta)\alpha$. We note that the composition of two substitutions is a substitution. To extend the preceding algorithm to literals L and M, return **fail** if L and M have different signs or predicate symbols. Suppose L and M both have the same sign and predicate symbol P. Suppose L and M are $P(r_1, r_2, \ldots, r_n)$ and $P(s_1, s_2, \ldots, s_n)$, respectively, or their negations. Then return **Unify_lists**$([r_1 \ldots r_n], [s_1 \ldots s_n])$ as the most general unifier of L and M.

As examples of unification, a most general unifier of the terms $f(x, a)$ and $f(b, y)$ is $\{x \mapsto b, y \mapsto a\}$. The terms $f(x, g(x))$ and $f(y, y)$ are not unifiable. A most general unifier of $f(x, y, g(y))$ and $f(z, h(z), w)$ is $\{x \mapsto z, y \mapsto h(z), w \mapsto g(h(z))\}$.

We can also define unifiers and most general unifiers of *sets* of terms. We say that a substitution α is a unifier of a set $\{t_1, t_2, \ldots, t_n\}$ of terms if $t_1\alpha \equiv t_2\alpha \equiv t_3\alpha \ldots$. If such a unifier α exists, we say that this set of terms is unifiable. It turns out that if $\{t_1, t_2, \ldots, t_n\}$ is a set of terms and has a unifier, then it has a most general unifier, and this can be computed as **Unify**$(f(t_1, t_2, \ldots, t_n), f(t_2, t_3, \ldots, t_n, t_1))$, where f is a function symbol of arity n. In a similar way, we can define most general unifiers of sets of literals.

Finally, suppose C_1 and C_2 are two clauses and A_1 and A_2 are nonempty subsets of C_1 and C_2, respectively. Suppose for convenience that there are no common variables between C_1 and C_2. Suppose the set $\{L : L \in A_1\} \cup \{\neg L : L \in A_2\}$ is unifiable, and let α be its most general unifier. We define the resolvent of C_1 and C_2 on the subsets A_1 and A_2 to be the clause

$$(C_1 - A_1)\alpha \cup (C_2 - A_2)\alpha$$

We define a resolvent of C_1 and C_2 to be a resolvent of C_1 and C_2 on two such sets A_1 and A_2 of literals. If C_1 and C_2 have common variables, we assume that the variables of one of these clauses are renamed before resolving to ensure that there are no common variables. There may be more than one resolvent of two clauses, or there may not be any resolvents at all.

Most of the time, A_1 and A_2 consist of single literals. This considerably simplifies the definition, and most of our examples will be of this special case. If $A_1 \equiv \{L\}$ and $A_2 \equiv \{M\}$, then we call L and M *literals of resolution.* We can call this kind of resolution *single literal resolution.* Often, we define resolution in terms of factoring and single literal resolution. If C is a clause and θ is a most general unifier of two distinct literals of C, then $C\theta$ is called a *factor* of C. Defining resolution in terms of factoring has some advantages, though it increases the number of clauses one must store.

Here are some examples. Suppose C_1 is $\{P(a)\}$ and C_2 is $\{\neg P(x), Q(f(x))\}$. Then a resolvent of these two clauses on the literals $P(a)$ and $\neg P(x)$ is $\{Q(f(a))\}$. This is because the most general unifier of these two literals is $\{x \mapsto a\}$, and applying this substitution to $\{Q(f(x))\}$ yields the clause $\{Q(f(a))\}$.

Suppose C_1 is $\{\neg P(a, x)\}$ and C_2 is $\{P(y, b)\}$. Then $\{\}$ (the empty clause) is a resolvent of C_1 and C_2 on the literals $\neg P(a, x)$ and $P(y, b)$.

Suppose C_1 is $\{\neg P(x), Q(f(x))\}$ and C_2 is $\{\neg Q(x), R(g(x))\}$. In this case, we rename the variables of C_2 first before resolving, to eliminate common variables. We obtain $\{\neg Q(y), R(g(y))\}$. Then a resolvent of C_1 and C_2 on the literals $Q(f(x))$ and $\neg Q(y)$ is $\{\neg P(x), R(g(f(x)))\}$.

Suppose C_1 is $\{P(x), P(y)\}$ and C_2 is $\{\neg P(z), Q(f(z))\}$. Then a resolvent of C_1 and C_2 on the sets $\{P(x), P(y)\}$ and $\{\neg P(z)\}$ is $\{Q(f(z))\}$.

A *resolution proof* of a clause C from a set S of clauses is a sequence $C_1, C_2, \ldots, C_n$ of clauses in which C_n is C and in which for all i, either C_i is an element of S or there exist integers $j, k < i$ such that C_i is a resolvent of C_j and C_k. Such a proof is called a (resolution) refutation from S if C_n is $\{\}$ (the empty clause).

A theorem proving method is called complete if it is able to prove any valid formula. For unsatisfiability testing, a theorem proving method is called complete if it can derive **false,** or the empty clause, from any unsatisfiable set of clauses. It is known that resolution is complete:

Theorem 4. A set S of first-order clauses is unsatisfiable iff there is a resolution refutation from S.

Therefore, we can use resolution to test the unsatisfiability of clause sets, and hence the validity of first-order formulas. The advantage of resolution over the **Prover** procedure is that resolution uses unification to choose instances of the clauses that are more likely to appear in a proof. So in order to show that a first-order formula A is valid, we can do the following:

- Convert $\neg A$ to clause form S.
- Look for a proof of the empty clause from S.

Here is an example of the whole procedure: Suppose that we want to show that the first-order formula $(\forall x \exists y\, (P(x) \supset Q(y))) \wedge (\forall y \exists z\, (Q(y) \supset R(z))) \supset (\forall x \exists z\, (P(x) \supset R(z)))$ is valid. Here $\supset$ represents logical implication, as usual. In the refutational approach, we negate this formula to obtain $\neg[(\forall x \exists y(P(x) \supset Q(y))) \wedge (\forall y \exists z(Q(y) \supset R(z))) \supset (\forall x \exists z(P(x) \supset R(z)))]$ and show that this formula is unsatisfiable. This is translated into clause form by rearranging the Boolean connectives and replacing existential quantifiers by new function symbols, called *Skolem functions.* By this means, we obtain the formula $(\forall x \exists y(P(x) \supset Q(y))) \wedge (\forall y \exists z(Q(y) \supset R(z))) \wedge (\exists x \forall z(P(x) \wedge \neg R(z)))$; that is, $(\forall x \exists y(P(x) \supset Q(y))) \wedge (\forall y \exists z(Q(y) \supset R(z))) \wedge (\exists x P(x)) \wedge \forall z \neg R(z)$. Inserting Skolem functions, we obtain $(\forall x(P(x) \supset Q(f(x)))) \wedge (\forall y(Q(y) \supset R(g(y)))) \wedge P(a) \wedge \forall z \neg R(z)$. This translation is satisfiability preserving. Translating this formula into a set S of clauses, we obtain $\{\{\neg P(x), Q(f(x))\}, \{\neg Q(y), R(g(y))\}, \{P(a)\}, \{\neg R(z)\}\}$. The variables are implicitly regarded as universally quantified. We then have the following resolution refutation:

1.	$P(a)$	(input)
2.	$\neg P(x), Q(f(x))$	(input)
3.	$Q(f(a))$	(1, 2, resolution)
4.	$\neg Q(y), R(g(y))$	(input)
5.	$R(g(f(a)))$	(3, 4, resolution)
6.	$\neg R(z)$	(input)
7.	**false**	(5, 6, resolution)

The designation "input" means that a clause is in S. Since **false** (the empty clause) has been derived from S by resolution, we have proven that S is unsatisfiable, and so the original first-order formula is valid.

Even though resolution is much more efficient than the **Prover** procedure, it is still not as efficient as we would like. In the early days of resolution, a number of refinements were added to resolution, mostly by the Argonne group, to make it more efficient. These were the *set of support* strategy, unit preference, hyper-resolution, subsumption and tautology deletion, and demodulation. In addition, the Argonne group preferred using small clauses when searching for resolution proofs. This has pretty much continued as their recipe until today, with the addition of some very efficient data structures for storing and accessing clauses. We will describe most of these refinements now.

A clause C is called a *tautology* if for some literal L, $L \in C$ and $\neg L \in C$. It is known that if S is unsatisfiable, there is a refutation from S that does not contain any tautologies. This means that tautologies can be deleted as soon as they are generated and need never be included in resolution proofs.

In general, given a set S of clauses, we search for a refutation from S by performing a sequence of resolutions. To ensure completeness, this search should be fair; that is, if clauses C_1 and C_2 have been generated already and it is possible to resolve these clauses, then this resolution must eventually be done. However, there is still considerable flexibility in which resolutions are done first, and a good choice in this respect can help the prover a lot. One good idea is to prefer resolutions of clauses that are small (that is, that have small terms in them).

Another idea is based on subsumption, as follows: We say that clause C *subsumes* D if there is a substitution Θ such that $C\Theta \subseteq D$. For example, the clause $\{Q(x)\}$ subsumes the clause $\{\neg P(a), Q(a)\}$. We say that C *properly subsumes* D if C subsumes D and the number of literals in C is less than or equal to the number of literals in D. For example, the clause $\{Q(x), Q(y)\}$ subsumes $\{Q(a)\}$ but does not properly subsume it. It is known that clauses properly subsumed by other clauses can be deleted when searching for resolution refutations from S. It is possible that these deleted clauses may still appear in the final refutation, but once a clause C is generated that properly subsumes D, it is never necessary to use D in any further resolutions. Subsumption deletion can reduce the proof time tremendously, since long clauses tend to be subsumed by short ones. Of course, if two clauses properly subsume each other, one of them should be kept.

We can put all of this together to give a program for searching for resolution proofs from S, as follows:

```
procedure Resolver(S)
  R ← S;
  while false ∉ R do
    choose clauses C1, C2 ∈ R fairly, preferring
      small clauses;
    if no new pairs C1, C2 exist then return
      ''satisfiable'' fi;
    R′ ← {D : D is a resolvent of C1, C2 and D is
      not a tautology};
    for D ∈ R′ do
      if no clause in R properly subsumes D
      then R ← {D} ∪ {C ∈ R : D does not properly
        subsume C} fi;
    od
  od
end Resolver
```

To make precise what a "small clause" is, we define $\|C\|$, the *symbol size* of clause C, as follows:

$$\|x\| = 1 \text{ for variables } x$$
$$\|c\| = 1 \text{ for constant symbols } c$$
$$\|f(t_1, \ldots, t_n)\| = 1 + \|t_1\| + \cdots + \|t_n\| \text{ for terms } f(t_1, \ldots, t_n)$$
$$\|P(t_1, \ldots, t_n)\| = 1 + \|t_1\| + \cdots + \|t_n\| \text{ for atoms } P(t_1, \ldots, t_n)$$
$$\|\neg A\| = \|A\| \text{ for atoms } A$$
$$\|\{L_1, L_2, \ldots, L_n\}\| = \|L_1\| + \cdots + \|L_n\| \text{ for clauses } \{L_1, L_2, \ldots, L_n\}$$

Small clauses, then, are those having a small symbol size.

We also give a program for testing if a clause C subsumes D:

```
procedure Subsumes(C, D)
  let x1, x2, . . ., xn be the variables in D;
  let c1, c2, . . ., cn be new constant symbols;
  let θ be {x1 ↦ c1, . . ., xn ↦ cn};
  return Subsumes2(C, Dθ);
end Subsumes

procedure Subsumes2(C, D)
  if C = {} then return true fi;
  let L be a literal in C;
    for literals M ∈ D do
      α ← Unify(L, M);
      if α ≠ fail and Subsumes2((C − {L})α, D)
        then return true fi;
    od
  return false;
end Subsumes2
```

The purpose of the substitution θ is to replace all variables of D by new constant symbols before calling **Subsumes2.** We note that **Subsumes**($\{L\}$, $\{M\}$) is **true** iff the literal M is an instance of L.

Another technique used by the Argonne group is the *unit preference strategy,* defined as follows: A *unit clause* is a clause that contains exactly one literal. A *unit resolution*is a resolution of clauses C_1 and C_2, where at least one of C_1 and C_2 is a unit clause. The unit preference strategy prefers unit resolutions when searching for proofs. Other resolutions are also performed, but not as early. The unit preference strategy helps because unit resolutions reduce the number of literals in a clause.

Demodulation is a way of replacing equals by equals, which permits simplification of expressions. We will discuss this later. Hyper-resolution is a refinement of resolution that restricts the inferences that are performed. Many such refinements have been developed, and we now discuss some of them.

Refinements of Resolution

In an attempt to make resolution more efficient, many refinements were developed in the early days of theorem proving. We present a few of them and mention a number of others. For a discussion of resolution and its refinements, and theorem proving in general, see (23–28). It is hard to know which refinements will help on any given example, but experience with a theorem prover can help to give one a better idea of which refinements to try. In general, none of these refinements helps very much most of the time.

A literal is called *positive* if it is an atom (that is, has no negation sign). A literal with a negation sign is called *negative.* A clause C is called positive if all of the literals in C are positive. C is called negative if all of the literals in C are negative. A resolution of C_1 and C_2 is called positive if one of C_1 and C_2 is a positive clause. It is called negative if one of C_1 and C_2 is a negative clause. It turns out that positive resolution is complete; that is, if S is unsatisfiable, then there is a refutation from S in which all of the resolutions are positive. This refinement of resolution is known as P_1 deduction in the literature. Similarly, negative resolution is complete. Hyper-resolution is essentially a modification of positive resolution in which a series of positive resolvents is done all at once. To be precise, suppose that C is a clause having at least one negative literal and $D_1, D_2, \ldots, D_n$ are positive clauses. Suppose C_1 is a resolvent of C and D_1, C_2 is a resolvent of C_1 and D_2, $\ldots$, C_n is a resolvent of C_{n-1} and D_n. Suppose that C_n is a positive clause, but none of the clauses C_i are positive, for $i < n$. Then C_n is called a *hyper-resolvent* of C and $D_1, D_2, \ldots, D_n$. Thus the inference steps in hyper-resolution are sequences of positive resolutions. Hyper-resolution is sometimes useful because it reduces the number of intermediate results that must be stored in the prover.

Typically, when proving a theorem, there is a general set A of axioms and a particular formula F that we wish to prove. So we wish to show that the formula $A \supset F$ is valid. In the refutational approach, we do this by showing that $\neg(A \supset F)$ is unsatisfiable. Now $\neg(A \supset F)$ is transformed to $A \wedge \neg F$ in the clause form translation. We then obtain a set S_A of clauses from A and a set S_F of clauses from $\neg F$. The set $S_A \cup S_F$ is unsatisfiable iff $A \supset F$ is valid. We typically try to show $S_A \cup S_F$ unsatisfiable by performing resolutions. Since we are attempting to prove F, we would expect that resolutions involving the clauses S_F are more likely to be useful, since resolutions involving two clauses from S_A are essentially combining general axioms. The set of support strategy is designed to force all resolutions to involve a clause in S_F or a clause derived from it.

Sets A of axioms typically have standard models I. Thus $I \models A$. Since clause form is satisfiability preserving, $I' \models S_A$ as well, where I' is obtained from I by a suitable interpretation of Skolem functions. The idea of the set of support strategy is to use some interpretation like I' to specify which clauses are relevant to the particular theorem; the relevant clauses are those that I' does not satisfy.

So, in general, the set of support strategy takes a set S of clauses and an interpretation I. We let T be the set of clauses C of S such that $I \not\models C$. Then T becomes the set of support, and it is required that all resolutions either involve a clause in T or a clause derived from T by other resolutions. It is known that the set of support strategy is complete.

Other refinements of resolution include *ordered resolution,* which orders the literals of a clause and requires that the subsets of resolution include a maximal literal in their respective clauses. Unit resolution requires all resolutions to be unit resolutions and is not complete. Input resolution requires all resolutions to involve a clause from S, and this is not complete, either. Unit resulting (UR) resolution is like unit resolution but has larger inference steps. This is also not complete but works well surprisingly often. Locking resolution attaches indices to literals and uses these to order the literals in a clause and decide which literals have to belong to the subsets of resolution. Ancestry-filter form resolution imposes a kind of linear format on resolution proofs. Semantic resolution is like set of support resolution but requires that when two clauses C_1 and C_2 resolve, at least one of them must not be satisfied by a specified interpretation I. These strategies are all complete. Semantic resolution is compatible with some ordering refinements (that is, the two strategies together are still complete).

It is interesting that resolution is complete for logical consequences in the following sense: If S is a set of clauses and C is a clause such that $S \models C$—that is, C is a logical consequence of S—then there is a clause D derivable by resolution from S such that D subsumes C.

Another resolution refinement that is sometimes useful is *splitting*. If C is a clause and $C \equiv C_1 \cup C_2$, where C_1 and C_2 have no common variables, then $S \cup \{C\}$ is unsatisfiable iff $S \cup \{C_1\}$ is unsatisfiable and $S \cup \{C_2\}$ is unsatisfiable. The effect of this is to reduce the problem of testing unsatisfiability of $S \cup \{C\}$ to two simpler problems. A typical example of such a clause C is a ground clause with two or more literals.

There is a special class of clauses, called *Horn clauses,* for which specialized theorem proving strategies are complete. A Horn clause is a clause that has at most one positive literal. Such clauses have found tremendous application in logic programming languages. If S is a set of Horn clauses, then unit resolution is complete, as is input resolution.

Other Strategies

There are a number of other strategies that apply to sets S of clauses but do not use resolution. One of the most notable is *model elimination* (4), which constructs chains of literals and has some similarities to the Davis and Putnam procedure. Model elimination also specifies the order in which literals of a clause will "resolve away." There are also a number of *connection methods* (29), which operate by constructing links between complementary literals in different clauses and creating structures containing more than one clause linked together. In addition, there are a number of *instance-based* strategies, which create a set T of ground instances of S and test T for unsatisfiability using a Davis and Putnam-like procedure. Such instance-based methods can be much more efficient than resolution on certain kinds of clause sets (namely, those that are highly non-Horn but do not involve deep term structure).

Furthermore, there are a number of strategies that do not use clause form at all. These include the semantic tableau methods, which work backward from a formula and construct a tree of possibilities; Andrews' matings method, which is suitable for second-order logic and has obtained some impressive proofs automatically; natural deduction methods; and sequent-style systems.

Evaluating Strategies

In general, qualities that need to be considered when evaluating a strategy are not only completeness but also propositional efficiency, goal sensitivity, and use of semantics. By propositional efficiency we mean how the efficiency of the method on propositional problems compares with Davis and Putnam's method; most strategies do poorly in this respect. By goal sensitivity, we mean the degree to which the method permits one to concentrate on inferences related to the particular clauses coming from the negation of the theorem (the set S_F discussed previously). When there are many input clauses, goal sensitivity is crucial. By use of semantics, we mean whether the method can take advantage of natural semantics that may be provided with the problem statement in its search for a proof. We note that model elimination and set of support strategies are goal sensitive but apparently not propositionally efficient. Semantic resolution is goal sensitive and can use natural semantics but is not propositionally efficient. Instance-based strategies are goal sensitive and use natural semantics and are propositionally efficient but sometimes have to resort to exhaustive enumeration instead of unification in order to instantiate clauses. A further issue is to what extent various methods permit the incorporation of efficient equality techniques, which varies a lot from method to method. So we see that there are some interesting problems involved in combining as many of these desirable features as possible. For strategies involving extensive human interaction, the criteria for evaluation are considerably different.

EQUALITY

When proving theorems involving equations, we obtain many irrelevant terms. For example, if we have the equations $x + 0 = x$ and $x * 1 = x$, and addition and multiplication are commutative and associative, then we obtain many terms identical to x, such as $1 * x * 1 * 1 + 0$. For products of two or three variables or constants, the situation becomes much worse. It is imperative to find a way to get rid of all of these equivalent terms. For this purpose, specialized methods have been developed to handle equality.

The most straightforward method of handling equality is to use a general first-order resolution theorem prover together with the *equality axioms,* which are the following (assuming free variables are implicitly universally quantified):

$$x = x$$

$$x = y \supset y = x$$

$$x = y \wedge y = z \supset x = z$$

$$x_1 = y_1 \wedge x_2 = y_2 \wedge \cdots \wedge x_n = y_n \supset f(x_1 \ldots x_n) = f(y_1 \ldots y_n)$$

$$\text{for all function symbols } f$$

$$x_1 = y_1 \wedge x_2 = y_2 \wedge \cdots \wedge x_n = y_n \wedge P(x_1 \ldots x_n) \supset P(y_1 \ldots y_n)$$

$$\text{for all predicate symbols } P$$

We let **Eq** refer to this set of equality axioms. The approach of using **Eq** explicitly leads to many inefficiencies, as noted previously, although in some cases it works reasonably well.

Another approach to equality is the *modification method* of Brand (30). In this approach, a set S of clauses is transformed into another set S' with the following property: $S \cup \mathbf{Eq}$ is unsatisfiable iff $S' \cup \{x = x\}$ is unsatisfiable. Thus this transformation avoids the need for the equality axioms, except for $\{x = x\}$. This approach often works a little better than using **Eq** explicitly.

Contexts

To discuss other inference rules for equality, we need some terminology. A *context* is a term with occurrences of $\square$ in it. For example, $f(\square, g(a, \square))$ is a context. A $\square$ by itself is also a context. We can also have literals and clauses with $\square$ in them, and they are also called contexts. If n is an integer, then an *n-context* is a term with n occurrences of $\square$. If t is an n-context and $m \leq n$, then $t[t_1, \ldots, t_m]$ represents t with the leftmost m occurrences of $\square$ replaced by the terms $t_1, \ldots, t_m$, respectively. Thus, for example, $f(\square, b, \square)$ is a 2-context, and $f(\square, b, \square)[g(c)]$ is $f(g(c), b, \square)$. Also, $f(\square, b, \square)[g(c)][a]$ is $f(g(c), b, a)$. In general, if r is an n-context and $m \leq n$ and the terms s_i are 0-contexts, then $r[s_1, \ldots, s_n] \equiv r[s_1][s_2] \ldots [s_n]$. However, $f(\square, b, \square)[g(\square)]$ is $f(g(\square), b, \square)$, so $f(\square, b, \square)[g(\square)][a]$ is $f(g(a), b, \square)$. In general, if r is a k-context for $k \geq 1$ and s is an n-context for $n \geq 1$, then $r[s][t] \equiv r[s[t]]$, by a simple argument (both replace the leftmost $\square$ in $r[s]$ by t).

Termination Orderings on Terms

We also need to discuss partial orderings on terms in order to explain inference rules for equality. A partial ordering $>$ is well founded if there are no infinite sequences x_i of elements such that $x_i > x_{i+1}$ for all $i \geq 0$. A *termination ordering* on terms is a partial ordering $>$ that is well founded and satisfies the *full invariance property*—that is, if $s > t$ and Θ is a substitution, then $s\Theta > t\Theta$—and also satisfies the *replacement property*—that is, $s > t$ implies $r[s] > r[t]$ for all 1-contexts r.

Note that if $s > t$ and $>$ is a termination ordering, then all variables in t appear also in s. For example, if $f(x) > g(x, y)$, then by full invariance $f(x) > g(x, f(x))$, and by replacement $g(x, f(x)) > g(x, g(x, f(x)))$, and so on, giving an infinite descending sequence of terms.

The concept of a *multiset* is often useful to show termination. Informally, a multiset is a set in which an element can occur more than once. Formally, a multiset S is a function from some underlying domain D to the nonnegative integers. It is said to be finite if $\{x : S(x) > 0\}$ is finite. We say $x \in S$ if $S(x) > 0$. We call $S(x)$ the *multiplicity* of x in S; this represents the number of times x appears in S. If S and T are multisets, then $S \cup T$ is defined by $(S \cup T)(x) = S(x) + T(x)$ for all x. A partial ordering $>$ on D can be extended to a partial ordering $\gg$ on multisets in the following way: We say $S \gg T$ if there is some multiset V such that $S = S' \cup V$ and $T = T' \cup V$ and for all t in T' there is an s in S' such that $s > t$. This relation can be computed reasonably fast by deleting common elements from S and T as long as possible, and then testing if the specified relation between S' and T' holds. The idea is that a multiset becomes smaller if an element is replaced by any number of smaller elements. Thus $\{3, 4, 4\} \gg \{2, 2, 2, 2, 1, 4, 4,\}$ since 3 has been replaced by 2, 2, 2, 2, 1. This operation can be repeated any number of times, still yielding a smaller multiset. We can show that if $>$ is well founded, so is $\gg$.

We now give some examples of termination orderings. The simplest kind of termination orderings are those that are based on size. Recall that $\|s\|$ is the symbol size (number of symbol occurrences) of a term s. We can then define $>$ so that $s > t$ if for all Θ making $s\Theta$ and $t\Theta$ ground terms, $\|s\Theta\| > \|t\Theta\|$. For example, $f(x, y) > g(y)$ in this ordering, but we do not have $h(x, a, b) > f(x, x)$ because x could be replaced by a large term. This termination ordering is computable; $s > t$ iff $\|s\| > \|t\|$ and no variable occurs more times in t than s.

We need more powerful techniques to get some more interesting termination orderings. One of the most remarkable results in this area is a theorem of Dershowitz (31) about simplification orderings, which gives a general technique for showing that an ordering is a termination ordering. Before his theorem, each ordering had to be shown well founded separately, and this was often difficult. First we define a simplification ordering.

Definition 1. A partial ordering $>$ on terms is a simplification ordering if it satisfies the replacement property—that is, for 1-contexts r, s, $> t$ implies $r[s] > r[t]$—and has the subterm property—that is, $s > t$ if t is a proper subterm of s. Also, if there are function symbols f with variable arity, we require that $f(\ldots s \ldots) > f(\ldots\ldots)$ for all such f.

Theorem 5. All simplification orderings are well founded.

Proof. Based on Kruskal's tree theorem (32), which says that in any infinite sequence $t_1 t_2 t_3 \ldots$ of terms, there is an i and j with $i < j$ such that t_i is embedded in t_j in a certain sense. It turns out that if t_i is embedded in t_j then $t_j \geq t_i$ for any simplification ordering $>$.

We now present the *recursive path ordering,* which is a simplification ordering. This ordering is defined in terms of a *precedence* ordering on function symbols, which is a partial ordering on the function symbols. We will say $f < g$ to indicate that f is less than g in the precedence relation on function symbols. We will present the recursive path ordering by a complete set of inference rules that may be used to construct proofs of $s > t$. That is, if $s > t$, then there is a proof of this in the system. Also, by using the inference rules backward in a goal-directed manner, it is possible to construct a reasonably efficient decision procedure for statements of the form $s > t$. Recall that if $>$ is an ordering, then $\gg$ is the extension of this ordering to multisets. The ordering we present is somewhat weaker than that usually given in the literature.

$$\frac{f = g \quad \{s_1 \ldots s_m\} \gg \{t_1 \ldots t_n\}}{f(s_1 \ldots s_m) > g(t_1 \ldots t_n)}$$

$$\frac{s_i \geq t}{f(s_1 \ldots s_m) > t}$$

$$\frac{\text{true}}{s \geq s}$$

$$\frac{f > g \quad f(s_1 \ldots s_m) > t_i \text{ all } i}{f(s_1 \ldots s_m) > g(t_1 \ldots t_n)}$$

For example, suppose $* > +$. Then we can show that $x * (y + z) > x * y + x * z$ as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{true}}{y+z > y}}{\{x, y+z\} \gg \{x, y\}}}{x*(y+z) > x*y} \quad \dfrac{\dfrac{\dfrac{\text{true}}{y+z > z}}{\{x, y+z\} \gg \{x, z\}}}{x*(y+z) > x*z} \quad * > +}{x*(y+z) > x*y + x*z}$$

For some purposes, it is necessary to modify this ordering so that subterms are considered lexicographically. In general, if $>$ is an ordering, then the lexicographic extension $>_{\text{lex}}$ of $>$ to tuples is defined as follows:

$$\frac{s_1 > t_1}{(s_1 \ldots s_m) >_{\text{lex}} (t_1 \ldots t_n)}$$

$$\frac{s_1 = t_1 \quad (s_2 \ldots s_m) >_{\text{lex}} (t_2 \ldots t_n)}{(s_1 \ldots s_m) >_{\text{lex}} (t_1 \ldots t_n)}$$

$$\frac{\text{true}}{(s_1 \ldots s_m) >_{\text{lex}} ()}$$

We can show that if $>$ is well founded, then so is its extension $>_{\text{lex}}$ to bounded length tuples. This lexicographic treatment of subterms is the idea of the lexicographic path ordering of Kamin and Levy. This ordering is defined by the following inference rules:

$$\frac{f = g \quad (s_1 \ldots s_m) >_{\text{lex}} (t_1 \ldots t_n) \quad f(s_1 \ldots s_m) > t_j, \text{ all } j \geq 2}{f(s_1 \ldots s_m) > g(t_1 \ldots t_n)}$$

$$\frac{s_i \geq t}{f(s_1 \ldots s_m) > t}$$

$$\frac{\text{true}}{s \geq s}$$

$$\frac{f > g \quad f(s_1 \ldots s_m) > t_i \text{ all } i}{f(s_1 \ldots s_m) > g(t_1 \ldots t_n)}$$

In the first inference rule, we do not need to test $s > t_1$ since $(s_1 \ . \ . \ . \ s_m) >_{\text{lex}} (t_1 \ . \ . \ . \ t_n)$ implies $s_1 \geq t_1$ and hence $s > t_1$. We can show that this ordering is a simplification ordering for systems having fixed arity function symbols. This ordering has the useful property that $f(f(x, y), z) >_{\text{lex}} f(x, f(y, z))$; informally, the reason for this is that the terms have the same size, but the first subterm $f(x, y)$ of $f(f(x, y), z)$ is always larger than the first subterm x of $f(x, f(y, z))$.

There are also many other orderings known that are similar to the preceding ones.

Paramodulation

Earlier, we saw that the equality axioms **Eq** can be used to prove theorems involving equality and that Brand's modification method is another approach that avoids the need for the equality axioms. A better approach in most cases is to use the *paramodulation rule,* defined as follows:

$$\frac{C[t], r = s \vee D, r \text{ and } t \text{ are unifiable}, \quad t \text{ is not a variable}, \mathbf{Unify}(r, t) = \theta}{C[s\theta] \vee D\theta}$$

Here $C[t]$ is a clause (1-context) C containing an occurrence of a nonvariable subterm t and $C[s\theta]$ is C with this occurrence of t replaced by $s\theta$. Also, $r = s \vee D$ is another clause having a literal $r = s$ whose predicate is equality and remaining literals D, which can be empty. To understand this rule, consider that $r\theta = s\theta$ is an instance of $r = s$, and $r\theta$ and $t\theta$ are identical. If $D\theta$ is false, then $r\theta = s\theta$ must be true, so we can replace $r\theta$ in C by $s\theta$ if $D\theta$ is false. Thus we infer $C[s\theta] \vee D\theta$. We assume, as usual, that variables in $C[t]$ or in $r = s \vee D$ are renamed if necessary to ensure that these clauses have no common variables before performing paramodulation. We say that the clause C is paramodulated *into.* We also allow paramodulation in the other direction—that is, the equation $r = s$ can be used in either direction.

For example, the clause $P(g(a)) \vee Q(b)$ is a paramodulant of $P(f(x))$ and $(f(a) = g(a)) \vee Q(b)$. Brand (30) showed that if **Eq** is the set of equality axioms given previously and S is a set of clauses, then $S \cup$ **Eq** is unsatisfiable iff there is a proof of the empty clause from $S \cup \{x = x\}$ using resolution and paramodulation as inference rules. Thus, paramodulation allows us to dispense with all the equality axioms except $x = x$. Some more recent proofs of the completeness of paramodulation (33) show the completeness of restricted versions of paramodulation that considerably reduce the search space. In particular, we can restrict this rule so that it is not performed if $s\theta > r\theta$, where $>$ is a termination ordering fixed in advance. So if we have an equation $r = s$, and $r > s$, then this equation can only be used to replace instances of r by instances of s. If $s > r$, then this equation can only be used in the reverse direction. The effect of this is to constrain paramodulation so that "big" terms are replaced by "smaller" ones, considerably improving its efficiency. It would be disaster if we allowed paramodulation to replace x by $x * 1$, for example, Another complete refinement of ordered paramodulation is that paramodulation only needs to be done into the "large" side of an equation. If the subterm t of $C[t]$ occurs in an equation $u = v$ or $v = u$ of C, and $u > v$, where $>$ is the termination ordering being used, then the paramodulation need not be done if the specified occurrence of t is in v. Some early versions of paramodulation required the use of the functionally reflexive axions of the form $f(x_1, \ . \ . \ ., x_n) = f(x1, \ . \ . \ ., x_n)$, but this is now known not to be necessary. When D is empty, paramodulation is similar to "narrowing," which has been much studied in the context of logic programming and term rewriting.

Demodulation

Similar to paramodulation is the rewriting or demodulation rule, which is essentially a method of simplification.

$$\frac{C[t], r = s, r\theta \equiv t, r\theta > s\theta}{C[s\theta]}$$

Here $C[t]$ is a clause (so C is a 1-context) containing a nonvariable term t, $r = s$ is a unit clause, and $>$ is the termination ordering that is fixed in advance. We assume that variables are renamed so that $C[t]$ and $r = s$ have no common variables before this rule is applied. We note that we can test if t is an instance of r, and obtain θ if so, by calling **Subsumes**($\{P(r)\}$, $\{P(t)\}$). We call $C[s\theta]$ a *demodulant* of $C[t]$ and $r = s$. Similarly, $C[s\theta]$ is a demodulant of $C[t]$ and $s = r$, if $r\theta > s\theta$. Thus an equation can be used in either direction if the ordering condition is satisfied.

As an example, if we have the equation $x * 1 = x$ and if $x * 1 > x$ and we have a clause $C[f(a) * 1]$ having a subterm of the form $f(a) * 1$, we can simplify this clause to $C[f(a)]$, replacing the occurrence of $f(a) * 1$ in C by $f(a)$.

To justify the demodulation rule, we can infer the instance $r\theta = s\theta$ of the equation $r = s$ because free variables are implicitly universally quantified. This permits us to replace $r\theta$ in C by $s\theta$, and vice versa. But $r\theta$ is t, so we can replace t by $s\theta$.

Not only is the demodulant $C[s\theta]$ inferred, but the original clause $C[t]$ is typically deleted. Thus, in contrast to resolution and paramodulation, demodulation replaces clauses by simpler clauses. This can be a considerable aid in reducing the number of generated clauses.

The reason for specifying that $s\theta$ is simpler than $r\theta$ is not only the intuitive desire to simplify clauses, but also to ensure that demodulation terminates. For example, we cannot have a termination ordering in which $x * y > y * x$, since then the clause $a * b = c$ could demodulate using the equation $x * y = y * z$ to $b * a = c$ and then to $a * b = c$, and so on indefinitely. Such an ordering $>$ could not be a termination ordering since it violates the well-foundedness condition. However, for many termination orderings $>$ we will have that $x * 1 > x$, and thus the clauses $P(x * 1)$ and $x * 1 = x$ have $P(x)$ as a demodulant if some such ordering is being used.

Ordered paramodulation is still complete if it and demodulation are done with respect to the same simplification ordering during the proof process. Demodulation is essential in practice, for without it we can generate expressions like $x * 1 * 1 * 1$ that clutter up the search space. Some complete refinements of paramodulation also restrict which literals can be paramodulated into, which must be the "largest" literals in the clause in a sense. Such refinements are typically used with resolution refinements that also restrict subsets of resolution to contain "large" literals in a clause. Another recent development is basic paramodulation, which restricts the positions in a term into which paramodulation can be done (34); this refinement was used in McCune's proof of the Robbins problem (8).

A different problem occurs with the associative-commutative axioms for a function f:

$$f(f(x, y), z) = f(x, f(y, z))$$
$$f(x, y) = f(y, x)$$

These axioms permit many different products of terms to be generated, and there is no simple way to eliminate any of them using a termination ordering. Many provers use associative-commutative (AC) unification instead (35), which builds these associative and commutative axioms into the unification algorithm. This can lead to powerful theorem provers, but it also causes a problem because the time to perform AC unification can be double exponential in the sizes of the terms being unified. Many other unification algorithms for other sets of equations have also been developed (36).

A beautiful theory of *term rewriting systems* has been developed to handle proofs involving *equational systems;* these are theorems of the form $E \supset e$, where E is a collection of equations and e is an equation. For such systems, term rewriting techniques often lead to very efficient proofs. The Robbins problem was of this form, for example. Term rewriting system-based provers essentially construct proofs by performing paramodulation and demodulation, applied to sets of equations. For a discussion of term rewriting techniques, see Refs. 37–39. It is also worth noting that some methods of proof by mathematical induction are based on the theory of term rewriting systems.

OTHER LOGICS

So far, we have considered theorem proving in general first-order logic. However, there are many more specialized logics for which more efficient methods exist. Examples include Presburger arithmetic, geometry theorems, inequalities involving real polynomials (for which Tarski gave a decision procedure), ground equalities and inequalities (for which congruence closure is an efficient decision procedure), modal logic, temporal logic, and many more specialized logics. Specialized logics are often built into provers or logic programming systems using constraints. Another specialized area is that of computing polynomial ideals, for which efficient methods have been developed.

Higher-Order Logic

In addition to the logics mentioned previously, there are more general logics to consider, including higher-order logics. Such logics permit quantification over functions and predicates as well as variables. The HOL prover uses higher-order logic and permits users to give considerable guidance in the search for a proof. Andrews's matings prover is more automatic and has obtained some impressive proofs fully automatically, including Cantor's theorem that the powerset of a set has more elements than the set. In general, higher-order logic often permits a more natural formulation of a theorem than first-order logic and shorter proofs, in addition to being more expressive. But the price is that the theorem prover is more complicated; in particular, higher-order unification is considerably more complex than first-order unification.

Mathematical Induction

Without going to a full higher-order logic, we can still obtain a considerable increase in power by adding mathematical induction to a first-order prover. The mathematical induction schema is the following one:

$$\frac{(\forall y)[[(\forall x)((x < y) \supset P(x))] \supset P(y)]}{(\forall y)P(y)}$$

Here $<$ is a well-founded ordering. Specializing this to the usual ordering on the integers, we obtain the following Peano induction schema:

$$\frac{P(0), (\forall x)(P(x) \supset P(x+1))}{(\forall x)P(x)}$$

With such inference rules, we can, for example, prove that addition and multiplication are associative and commutative, given their straightforward definitions. Both of these induction schemas are second order, because the predicate P is implicitly universally quantified. The problem in using these schemas in an automatic theorem prover is in instantiating P. Once this is done, the induction schema can often be proved by first-order techniques. In fact, this is one way to adapt a first-order prover to perform mathematical induction—that is, to permit a human to instantiate P.

By instantiating P, we mean replacing $P(y)$ in the preceding formula by $A[y]$ for some first-order formula A containing the variable y. Equivalently, this means instantiating P to the function $\lambda z.A[z]$. When we do this, the first of the preceding schemes becomes

$$\frac{(\forall y)[[(\forall x)((x < y) \supset A[x])] \supset A[y]]}{(\forall y)A[y]}$$

We note that the hypothesis and conclusion are now first-order formulas. This instantiated induction schema can then be given to a first-order prover. One way to do this is to have the prover prove the formula $(\forall y)[[(\forall x)((x < y) \supset A[x])] \supset A[y]]$ and then conclude $(\forall y)A[y]$. Another approach is to add the first order formula $\{(\forall y)[[(\forall x)((x < y) \supset A[x])] \supset A[y]]\} \supset \{(\forall y)A[y]\}$ to the set of axioms. Both approaches are facilitated by using a structure-preserving translation of these formulas to clause form, in which the formula $A[y]$ is defined to be equivalent to $Q(y)$ for a new predicate symbol Q.

A number of semiautomatic techniques for finding such a formula A and choosing the ordering $<$ have been developed. One of them is the following: To prove that for all finite ground terms t, $A[t]$, first prove $A[c]$ for all constant symbols c, and then for each function symbol of of arity n prove that $A[t_1] \wedge A[t_2] \wedge \cdots \wedge A[t_n] \supset A[f(t_1, t_2, \ldots, t_n)]$. This is known as *structural induction* and is often reasonably effective.

A common case when an induction proof may be necessary is when the prover is not able to prove the formula $(\forall x)A[x]$, but the formulas $A[t]$ are separately provable for all ground terms t. Analogously, we may not be able to prove that $(\forall x)(\text{natural\_number}(x) \supset A[x])$, but we may be able to prove $A[0], A[1], A[2], \ldots$ individually. In such a case, it is reasonable to try to prove $(\forall x)A[x]$ by induction, instantiating $P(x)$ in the preceding schema to $A[x]$. However, this still does not specify which ordering $<$ to use. For this, it can be useful to detect how long it takes to prove the $A[t]$ individually. For example, if the time to prove $A[n]$ for natural number n is proportional to n, then we may want to try the usual (size) ordering on natural numbers. If $A[n]$ is easy to prove for all even n but for odd n the time is proportional to n, then we may try to prove the even case directly without induction and the odd case by induction, using the usual ordering on natural numbers.

The Boyer–Moore prover (6) has mathematical induction techniques built in, and many difficult proofs have been done on it, generally with substantial human guidance. A number of other provers also have automatic or semiautomatic induction proof techniques.

Set Theory

Since most of mathematics can be expressed in terms of set theory, it is logical to develop theorem proving methods that apply directly to theorems expressed in set theory. Second-order provers do this implicitly. First-order provers can be used for set theory as well; Zermelo–Fraenkel set theory consists of an infinite set of first-order axioms, and so we again have the problem of instantiating the axiom schemas so that a first-order prover can be used. There is another version of set theory known as von Neumann–Bernays–Gödel set theory, which is already expressed in first-order logic. Quite a bit of work has been done on this version of set theory as applied to automated deduction problems. Unfortunately, this version of set theory is somewhat cumbersome for a human or for a machine. Still, some mathematicians have an interest in this approach. There are also a number of systems in which humans can construct proofs in set theory, such as Mizar (15) and others. In fact, there is an entire project (the QED project) devoted to formalizing mathematics (40).

It is interesting to note in this respect that many set theory proofs that are simple for a human are very hard for resolution and other clause-based theorem provers. This includes theorems about the associativity of union and intersection. In this area, it seems worthwhile to incorporate more of the simple definitional replacement approaches used by humans into clause-based theorem provers.

As an example of the problem, suppose that we desire to prove that $(\forall x)((x \cap x) = x)$ from the axioms of set theory. A human would typically prove this by noting that $(x \cap x) = x$ is equivalent to $((x \cap x) \subset x) \wedge (x \subset (x \cap x))$, then observing that $A \subset B$ is equivalent to $(\forall y)((y \in A) \supset (y \in B))$, and finally observing that $y \in (x \cap x)$ is equivalent to $(y \in x) \wedge (y \in x)$. After applying all of these equivalences to the original theorem, a human would observe that the result is a tautology, thus proving the theorem.

But for a resolution theorem prover, the situation is not so simple. The axioms needed for this proof are

$$\begin{aligned}(x = y) &\equiv [(x \subset y) \wedge (y \supset x)]\\ (x \subset y) &\equiv (\forall z)((z \in x) \supset (z \in y))\\ (z \in (x \cap y)) &\equiv [(z \in x) \wedge (z \in y)]\end{aligned}$$

When these are all translated into clause form and Skolemized, the intuition of replacing a formula by its definition gets lost in a mass of Skolem functions, and a resolution prover has a much harder time. This example may be easy enough for a resolution prover to obtain, but other examples that are easy for a human quickly become very difficult for a resolution theorem prover using the standard approach.

The problem is more general than set theory and has to do with how definitions are treated by resolution theorem provers. One possible method to deal with this problem is to use "replacement rules," as described in (41). This gives a considerable improvement in efficiency on many problems of this kind.

CURRENT RESEARCH AREAS

We only have space to mention some of the major research areas in automatic theorem proving; in general, research is being conducted in all the areas described so far. Probably theorem provers are already more powerful than most people realize, although they are far from the level of performance we would like.

There is a continued development of new resolution strategies and other theorem proving techniques, such as instance-based methods. New methods for incorporating semantics into theorem provers are being developed. Proof planning is being studied as a way to enable humans better to guide the proof process. Structured editors and techniques for presenting and editing proofs are under development. There is also interest in methods of making machine-generated proofs easier for humans to understand. Development of more efficient data structures and the utilization of concurrency promise a continued increase in power for theorem provers.

One technique that can improve the efficiency of a theorem prover substantially is the use of *sorts,* and this is the subject of investigation. When there are many axioms, we have the problem of deciding which ones are relevant, and techniques for solving this problem (*gazing*) are being developed. Abstraction and analogy are being studied as aids in finding proofs faster. The idea is that if two problems are similar, then a proof for one of them may be useful in guiding the search for a proof for the other one.

Mathematical induction is another active area of research, since so many theorems require some kind of induction. There is also substantial interest in theorem proving in set theory and higher-order logic.

Another area of research is that of analyzing the complexity of theorem proving strategies, which gives a machine-independent estimate of their efficiency. This can be done in terms of proof length or search space size (number of clauses generated).

There are many specialized logics that are receiving attention. There is continued study into methods for theorem proving in nonclassical logics such as intuitionistic logic and temporal and modal logic. Better propositional decision procedures are being developed. Additional specialized decision procedures for other logics are being studied. The question of how to combine specialized decision procedures and how to incorporate them into a general first-order or higher-order theorem prover is of interest. It is interesting that in some cases, resolution theorem proving strategies are decision procedures for special classes of first-order formulas (42), and this area is receiving continued attention. Many areas of term rewriting systems are also subjects of current research, including development of new termination orderings, new complete refinements of paramodulation and demodulation, and new E-unification algorithms.

Many application areas are receiving attention, spurred on by the need for reliable software in critical applications. Even though theorem provers can be very complex programs, proof checkers tend to be quite simple, and we can gain confidence in the correctness of machine-generated proofs by running them through a number of proof checkers written in several languages and running on several machines. Hardware verification is an important application area that receives continued study. Program verification and program generation are also of interest. Provers are also being used by mathematicians as aids in their research, in some cases.

In closing, we would like to apologize for neglecting the many nonresolution strategies which have been developed, particularly in Europe. It was not possible to cover everything at the same level as resolution in the space provided. The plan of this survey was to give a relatively thorough treatment of one approach, namely resolution, and to make this treatment accessible to those with little background in this area. However, it is no longer true that resolution and paramodulation dominate all other theorem proving methods. Other approaches to theorem proving have significant advantages over resolution on certain kinds of problems. For example, SETHEO and Gandalf, the winners of the international systems competitions during the theorem proving conferences CADE-13 in 1996 and CADE-14 in 1997, are both not based on a saturation method like resolution. We also want to call attention to leading systems like SPASS, developed by Weidenbach at the Max-Planck Institute in Saarbruecken, Germany, and other systems, and the methods underlying them.

Furthermore, the work of many individuals (such as Woody Bledsoe) was not mentioned, and we apologize for this. It was also not possible to mention all relevant research areas. Despite this, we hope that this brief survey will at least give a flavor of the substantial activity in this fascinating area of human endeavor.

ACKNOWLEDGMENT

This research was partially supported by the National Science Foundation under grant CCR-9627316.

BIBLIOGRAPHY

1. P. C. Gilmore, A proof method for quantification theory, *IBM J. Res. Dev.,* **4**: 28–35, 1960.
2. J. Robinson, Theorem proving on the computer, *J. Assoc. Comput. Mach.,* **10**: 163–174, 1963.
3. J. Robinson, A machine-oriented logic based on the resolution principle, *J. Assoc. Comput. Mach.,* **12**: 23–41, 1965.
4. D. Loveland, A simplified format for the model elimination procedure, *J. ACM,* **16**: 349–363, 1969.
5. P. B. Andrews, Theorem proving via general matings, *J. Assoc. Comput. Mach.,* **28**: 193–214, 1981.
6. R. Boyer and J. Moore, *A Computational Logic,* New York: Academic Press, 1979.
7. W. McCune, Otter 2.0 (theorem prover), in M. E. Stickel (ed.), *Proc. 10th Int. Conf. Automated Deduction,* 1990, pp. 663–664.
8. W. W. McCune, Solution of the Robbins problem, *J. Automated Reasoning,* **19** (3): 263–276, 1997.
9. D. Mitchell, B. Selman, and H. Levesque, Hard and easy distributions of SAT problems, *Proc. 10th Natl. Conf. Artificial Intell. (AAAI-92),* 1992, pp. 459–465.
10. R. Bryant, Symbolic Boolean manipulation with ordered binary-decision diagrams, *ACM Comput. Surveys,* **24** (3): 293–318, 1992.
11. J. Burch et al., Symbolic model checking: 10^{20} states and beyond, *Information Computat.,* **98**: 142–170, 1992.
12. D. Loveland, *Automated deduction: Some achievements and future directions,* technical report, Duke University, 1997.
13. M. J. Gordon and T. F. Melham (eds.), *Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic,* Cambridge, UK: Cambridge University Press, 1993.
14. L. C. Paulson, *Isabelle: A Generic Theorem Prover,* LNCS, vol. 828, New York: Springer-Verlag, 1994.
15. A. Trybulec and H. Blair, Computer aided reasoning with Mizar, in R. Parikh (ed.), *Logic of Programs,* LNCS, vol. 193, New York: Springer-Verlag, 1985.
16. R. L. Constable et al., *Implementing Mathematics with the Nuprl Proof Development System,* Englewood Cliffs, NJ: Prentice-Hall, 1986.
17. S. Owrie, J. M. Rushby, and N. Shankar, PVS: A prototype verification system, in D. Kapur (ed.), *Proc. 11th Conf. Automated Deduction,* June 1992, pp. 748–752. Lect. Notes in Artificial Intell. 607.
18. C. B. Suttner and G. Sutcliffe, *The TPTP problem library (TPTP v2.0.0),* Technical Report AR-97-01, Institut für Informatik, Technische Universität München, Germany, 1997.
19. D. Plaisted and Y. Zhu, *The Efficiency of Theorem Proving Strategies: A Comparative and Asymptotic Analysis,* Wiesbaden: Vieweg, 1997.

20. M. Davis, G. Logemann, and D. Loveland, A machine program for theorem-proving, *Commun. ACM,* **5**: 394–397, 1962.
21. M. Davis and H. Putnam, A computing procedure for quantification theory, *J. Assoc. Comput. Mach.,* **7**: 201–215, 1960.
22. A. Haken, The intractability of resolution, *Theoretical Comput. Sci.,* **39**: 297–308, 1985.
23. M. Fitting, *First-Order Logic and Automated Theorem Proving,* New York: Springer-Verlag, 1990.
24. C. Chang and R. Lee, *Symbolic Logic and Mechanical Theorem Proving,* New York: Academic Press, 1973.
25. D. Loveland, *Automated Theorem Proving: A Logical Basis,* New York: North-Holland, 1978.
26. A. Bundy, *The Computer Modelling of Mathematical Reasoning,* New York: Academic Press, 1983.
27. L. Wos et al., *Automated Reasoning: Introduction and Applications,* Englewood Cliffs, NJ: Prentice-Hall, 1984.
28. A. Leitsch, *The Resolution Calculus,* Berlin: Springer-Verlag, 1997. Texts in Theoretical Computer Science.
29. W. Bibel, *Automated Theorem Proving,* 2nd ed., Braunschweig/Wiesbaden: Vieweg, 1987.
30. D. Brand, Proving theorems with the modification method, *SIAM J. Comput.,* **4**: 412–430, 1975.
31. N. Dershowitz, Termination of rewriting, *J. Symbolic Comput.,* **3**: 69–116, 1987.
32. J. B. Kruskal, Well-quasi-ordering, the tree theorem, and Vazsonyi's conjecture, *Trans. AMS,* **95**: 210–225,1960.
33. J. Hsiang and M. Rusinowitch, Proving refutational completeness of theorem-proving strategies: The transfinite semantic tree method, *J. Assoc. Comput. Mach.,* **38** (3): 559–587, 1991.
34. Leo Bachmair et al., Basic paramodulation, *Inf. Comput.,* **121** (2): 172–192, 1995.
35. M. E. Stickel, A unification algorithm for associative-commutative functions, *J. Assoc. Comput. Mach.,* **28**: 423–434, 1981.
36. J. Siekmann, Unification theory, *J. Symbolic Computat.,* **7**: 207–274, 1989.
37. N. Dershowitz and J.-P. Jouannaud, Rewrite systems, in J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science,* Amsterdam: North-Holland, 1990.
38. D. Plaisted, Equational reasoning and term rewriting systems, in D. Gabbay et al. (eds.), *Handbook of Logic in Artificial Intelligence and Logic Programming,* vol. 1, Oxford: Oxford University Press, 1993, pp. 273–364.
39. Jan Willem Klop, Term rewriting systems, in S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum (eds.), *Handbook of Logic in Computer Science,* vol. 2, Oxford: Oxford University Press, 1992, pp. 1–117.
40. QED Group, The QED manifesto, in A. Bundy (ed.), *Proc. 12th Int. Conf. Automated Deduction,* New York: Springer-Verlag, 1994, pp. 238–251. Lect. Notes Artificial Intell. 814.
41. S.-J. Lee and D. Plaisted, Use of replace rules in theorem proving, *Meth. Logic Comput. Sci.,* **1**: 217–240, 1994.
42. W. H. Joyner, Resolution strategies as decision procedures, *J. ACM,* **23** (1): 398–417, 1976.

DAVID A. PLAISTED
University of North Carolina at Chapel Hill

THEOREMS, NETWORK. See NETWORK THEOREMS.