AUTOMATED THEOREM PROVING AND
LOGIC PROGRAMMING: A NATURAL SYMBIOSIS*†

LARRY WOS AND WILLIAM McCUNE

We present a detailed review of the elements of automated theorem proving, emphasizing certain aspects of especial interest to the logic programming community. In particular, we focus heavily on how an automated theorem-proving program can treat equality in a natural and yet effective manner, and how such a program can use strategy to control its reasoning in a sophisticated fashion. With the objective of significantly increasing the scope of logic programming, perhaps some unusually inventive researcher can adapt various procedures we review in this article, and adapt them in a way that preserves most of the speed offered by logic programming. In turn, although our expertise rests far more in automated theorem proving, we include certain observations concerning the value of logic programming to automated theorem proving in general. In other words, a natural symbiosis between automated theorem proving and logic programming exists, which nicely completes the circle, since logic programming was born of automated theorem proving.

1. MOTIVATION

Occasionally one discovers that two fields, with apparently widely separate aims, in fact offer each other useful elements of theory and powerful techniques of implementation. In other words, a natural symbiosis exists. The two fields we have in mind are automated theorem proving and logic programming.

*Invited paper.
†This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/91/$3.50
At the most general level, the natural symbiosis between automated theorem proving and logic programming rests with three factors.

(1) Historically, the study of logic programming stems from the study of automated theorem proving, especially from the research of the early 1960s.

(2) After approximately twenty-five years, automated theorem proving can again play an important role for logic programming by offering what is needed to address two important aspects, that focusing on the effective treatment of equality, and that focusing on the sophisticated use of a variety of strategies.

(3) Logic programming can in turn offer to automated theorem proving implementation techniques to sharply reduce the CPU time required to complete assignments.

Although we leave a direct treatment of the first factor to various authors (see, in particular, the informative article by Elcock [9]), we do provide in Section 2 a review of automated theorem proving. A reading of the review may complete one's understanding of the roots of logic programming. Indeed, with binary resolution [26] (Section 2.2.1) as the basis, logic programming in the obvious sense is born of theorem proving.

Of more immediate and practical value is our treatment of the second and third factors just cited. In particular, we provide solutions for the problems of how to effectively cope with equality (Sections 2.2.2 and 2.4) and how to materially broaden the scope of logic programming by using more powerful strategies (Section 2.3). To complete the picture, we discuss in some detail where the advances in logic programming will benefit automated theorem proving. In short, we show why the term symbiosis is appropriate and why—for some—symbiosis is the perfect term.

Both automated theorem proving and logic programming are empirical sciences, equally dependent on theory, implementation, and application. In addition, and perhaps more important, both fields are primarily interested in logical reasoning. And yet, from a global perspective, an essential difference exists between the two fields (see Section 3). Automated theorem proving is oriented to logical reasoning that might be termed unfocused and unalgorithmic— to borrow from our colleague Ross Overbeek—while logic programming is oriented toward focused and algorithmic reasoning. Indeed, the former is concerned with general-purpose and often deep reasoning systems, while the latter is concerned (with certain exceptions) with programming languages and far less deep reasoning. Among the exceptions are the applications of logic programming to deductive databases, expert systems, and theorem proving—applications that obviously can require deep reasoning.

Especially, but not exclusively, for researchers interested in such applications, we shall show why the use of some version of the procedures that are so vital to automated theorem proving and that we discuss in Section 2 is indispensable. Indeed, the greater the depth of reasoning that is desired, the greater the need for the use of techniques like those employed in automated theorem proving.

Of course, logic programming does offer excellent features for symbolic manipulation, but—and here we have an issue that we address more fully in Sections 2
and 5—far more is needed to provide a firm foundation on which to build a
deductive system for first-order logic, with or without equality. One of the most
important features that are needed for effective automated theorem proving but
that are lacking in logic programming as currently defined is the heavy use of
different types of strategy, a fact that is often overlooked by researchers who wish
to rely almost solely on logic programming for building a powerful theorem prover.

Despite these differences, each field offers the other important benefits (Section
4). To see that this view is shared by members of the logic programming commu-
nity, we quote from F. Pereira's article [23]:

As researchers in automated deduction come up with new deductive procedures and
strategies that happen to fit the requirements of logic programming, in equality reasoning
for example, the generality of the idea of logic programming is widened; and conversely,
developments in logic programming—as, for example, the efficient implementation tech-
niques for Prolog—can be generalized to help automated deduction.

The latter has already occurred, as evidenced in the work of Overbeek and his
colleagues [5] and the work of Stickel [29]; the time for the former to occur is now.

Appropriately, our explicit objective is to encourage researchers to adapt
features from automated theorem proving to extend the usefulness and increase
the power of logic programming. Among the features to adapt are an effective
built-in treatment of equality and, more important, a sophisticated use of semanti-
cally oriented strategy. Indeed, that is the nature of the suggestions we make
throughout this article—to add to, not to replace, the features of logic program-
ing. Of course, because we are still fascinated with automated theorem proving
even after more than twenty-five years of research, we also have the implicit and
not so secret goal of acquainting (or, for those who already have some knowledge,
more fully acquainting) members of the logic programming community with the
elements of automated theorem proving.

The main reason for pursuing the explicit objective of encouraging the recom-
manded research rests with the recognition that the adaptation of procedures from
one field to another often results in important insights, extensions, and improve-
ments for both fields. Should such occur, we would have a beautiful example of
ideas borrowed by one field from another, followed by a return with interest—from
automated theorem proving to logic programming and back. The benefits for the
logic programmer in studying the elements of automated theorem proving rest with
the insight gained from reviewing the historical connection of the two fields and
with the increased familiarity with (possibly new) concepts that merit adaptation.

With regard to how one might make the desired adaptations, we can only
suggest the merest beginning. Specifically, one might begin by closely examining
the elements that appear to provide the keys to the successes for automated
theorem proving, at least the successes reported here (Section 1.2.1). One of the
keys to much of the success of the researchers at Argonne National Laboratory
rests with the natural and yet effective way in which equality is treated. Part of that
treatment rests with the use of an inference rule (paramodulation [25,39]) that
enables a reasoning program to treat equality as if it were "understood". By using
paramodulation, a program can reason directly deep within an expression, making
the desired substitutions in a single step without need of the usual axioms for
equality (Section 2.2.2). Perhaps some enterprising researcher can extend logic
programming by adapting paramodulation in an effective manner.
For a second example of how logic programming might be extended, perhaps a way can be found to use demodulation [41], a procedure for automatically canonicalizing expressions by applying the appropriate equalities as rewrite rules. The use of demodulation, like the use of paramodulation, sharply increases the power of automated theorem-proving programs. Their use by logic programming might markedly increase the usefulness of the corresponding programs. In Sections 2.2.2 and 2.4, we provide the details to enable one to understand more fully why our built-in treatment of equality affects the performance of a theorem-proving program so dramatically.

With regard to how one might begin the adaptation of the types of strategy used in automated theorem proving—strategy to restrict and also strategy to direct a program's reasoning—one might start by examining the material of Section 2.3, where we discuss the current use by logic programming of a limited form of the set of support strategy [40]. That the set of support strategy, which plays such a vital role in automated theorem proving, also plays an important role in logic programming may come as a surprise to many researchers. To aid the suggested research, we also discuss in Section 2.3 how the use of this semantically oriented strategy might be profitably extended. By extending the way in which the set of support strategy is used, logic programming can offer the user the opportunity of having the corresponding program reason recursively from just those statements that should be the focus of attention. Even more than an effective treatment of equality, a greatly expanded use of strategy will materially increase the power and usefulness of logic programming.

If, in addition, someone finds a means to give logic programming access to the weighting strategy [17] (Section 2.3 below) for directing the reasoning and access to subsumption [26] (Section 2.5 below) for substantially reducing the paths to explore in search of a desired answer, then logic programming will have taken advantage of much of what automated theorem proving has to offer. On the other hand, without employing strikingly more powerful strategy than that typically found in logic programming, hard problems, especially interesting theorems from mathematics and logic, will in general be out of reach. In particular, to have the power to prove even fairly difficult theorems, an automated theorem-proving program based on logic programming will require the incorporation of semantically oriented strategy.

Postponing the presentation of the details and the discussion of additional suggestions to later sections, let us immediately focus on our qualifications and general outlook.

1.1. Our General Outlook

To remove all doubts regarding our placement of automated theorem proving before logic programming in the title of this article, we note that our expertise lies in the field of automated theorem proving and not in logic programming. Indeed, the serious study at Argonne National Laboratory of the possibility of automating theorem proving began in 1963, and has continued without a break since then. We are deeply engrossed in that study and, as is so evident throughout this article, fascinated with the prospects and results. Therefore—predictably—this article will feature automated theorem proving far more than it does logic programming. Nevertheless, from various discussions with colleagues, we have learned enough to
permit us to comment on both fields, and to focus on the potential value each has for the other.

We view logic programming as "the study and use of carefully controlled deductive procedures as computing mechanisms" [23]. We also agree with F. Pereira [22] that the restriction to Horn clauses (clauses containing at most one positive literal) can be worse than an inconvenience. Fortunately, the recognition of that fact by various members of the logic-programming community has in fact led to studies focusing on dialects that are not restricted to the use of Horn clauses. Nevertheless, although we do not consider logic programming as identified with any of its incarnations such as PROLOG, many of our remarks are oriented toward PROLOG and PROLOG-like systems. Similarly, although we do not consider all of automated theorem proving to be identified with the approach taken at Argonne National Laboratory, nevertheless many of our remarks are oriented toward automated theorem-proving programs designed and implemented by researchers at this laboratory: programs such as AURA [28], iTP [15, 16], and OTTER [18].

Despite the successes of and our attachment to the Argonne approach, we openly recognize various formidable obstacles to overcome and research problems to solve [35]. A strategy to sharply curtail the fecundity of paramodulation is needed. A strategy that recursively extends the power of the set of support strategy would be welcomed. A more effective treatment of set theory is required. We have virtually no idea of what to do about definition expansion and contraction. Our programs as yet do not offer induction.

More general, and common to the cited obstacles and others not mentioned here, is the obstacle of CPU time. The recognition of this obstacle of execution speed is, of course, one of the key factors in motivating various studies in logic programming. In particular, the desire to sharply reduce this obstacle may be the main cause for the interest in theorem-proving programs based on logic-programming technology. Unfortunately, that approach to the automation of theorem proving merely replaces one obstacle, CPU time, by another, inaccessibility to retained information. Of course, one might say that such inaccessibility is not an obstacle; rather, it is the key to the power offered by logic programming. We shall discuss in Section 2 why such an assertion misses the point, why the retention of information is essential—at least when the application focuses on solving hard problems or proving even moderately interesting theorems. For such applications, the need for information retention rests in part with the fact that a program cannot afford to continually rediscover key lemmas, facts, and relationships.

For a taste of what is to be discussed regarding this need, we note that, without the retention of some intermediate information, an attempt at solving a hard problem will ordinarily lead to the pursuit of an inordinately large number of unprofitable paths. In contrast, the retention of information gives the program access to the use of demodulation, subsumption, and powerful strategies, with the result that many of those unprofitable paths are never considered. Naturally, the key question to answer focuses on the type of information to retain, which is one of the reasons we discuss in Section 2.2.3 linked inference and its possible implementation based on adapting techniques from logic programming.

Summarizing, especially (but not exclusively) if the goal is to rely on logic programming to solve even harder problems, one might study our case for information retention, our treatment of equality, and, most important, our use of
strategy. Such a study, even if not completely persuasive, may lead to the insights needed to extend logic programming in important ways. Of course, we believe we have found the best path to pursue, no doubt greatly influenced by our successful use of various automated theorem-proving programs. Even if we are in error, we can quickly and easily present a number of causes for optimism.

1.2. Causes for Optimism

To set the stage for the material covered in the succeeding sections, let us briefly present our current position and prepare one for the obvious optimism (and sometimes wild excitement) that pervades this article. At the top level, although other factors exist, our optimism rest chiefly on

1. the successes—by relying heavily on a computer program that reasons logically—with answering open questions and verifying computer programs;
2. the availability of portable theorem-proving programs—some of which are in the public domain—that offer impressive power and, if run on a personal computer, offer enough power to solve small problems;
3. the sharp increase in the willingness—often eagerness—of researchers to experiment; and
4. access to a variety of computers that rely on parallel processing, and the significance of the advances occurring now in automated theorem proving and in logic programming.

Keeping in mind that advances in both automated theorem proving and logic programming rely equally on developments in theory, implementation, and application, let us immediately focus, in the order given, on the four causes for our optimism and excitement.

1.2.1. Answering Open Questions and Verifying Computer Programs. Rather than a full understanding of the various successes, the nature of each achievement is the key for this diverse audience. Therefore, since the details are available elsewhere, we shall—in all but the case focusing on combinatory logic—simply touch on the open questions that have been answered and the computer programs that have been verified.

The significance of answering open questions with the assistance of an automated theorem-proving program rests with the fact that, ordinarily, such questions can be answered only by an expert. Therefore, when we can demonstrate that a computer program played an important role, we have strong evidence that such a program can reason—not only logically, but very effectively. The achievement becomes even more impressive when one realizes that so eminent a logician as Łukasiewicz asserted in 1948 that a formalized proof cannot be “discovered mechanically”, but can only be “checked mechanically” [14].

In contrast, the significance of verifying computer programs with the aid of an automated theorem-proving program rests with the importance of knowing that computer programs do in fact perform as they are intended to perform. Many well-known researchers still consider this activity—despite the existing achievements—to be essentially doomed. In fact, many well-known researchers consider
the activity of proving theorems with a computer also to be doomed. The next few years will prove both views to be even more in error. We say “even more in error” because of the following evidence.

In 1967, an automated theorem-proving program made its first contribution to mathematics: SAM’s lemma was proved. Next, although the effort was spread over an entire decade (starting in 1977), a number of open questions were answered, and answered with substantial assistance from an automated theorem-proving program. The first of those questions was taken from ternary Boolean algebra. The question focuses on the possible independence of each of the first three axioms for such an algebra. This question offered an important degree of piquancy in that it was posed by A. A. Grau, who invented ternary Boolean algebra. With S. Winker’s formulation of a technique for generating models with a theorem-proving program [31], proofs of the independence of each of the three axioms were obtained.

Of those answered in the period beginning in 1977, the second open question focuses on finite semigroups. The question, posed by the algebraist I. J. Kaplansky, asks about the existence of a finite semigroup that admits, in a nontrivial way, one type of mapping but avoids another type. By finding a semigroup of order 83 (containing 83 elements) a theorem-proving program answered the question in the affirmative. To do so, the program was instructed to cope with the problem of studying generators and relations [33], rather than attempting to employ a brute-force method. A brute-force approach was untenable because there exist more than 800,000 semigroups of order 7. The approach that succeeded did not take advantage of any expertise in the particular area of mathematics from which the problem was selected; the researchers studying the problem had none.

The third open question was taken from an area of logic known as equivalential calculus. The question, brought to our attention by the logician J. Kalman, asks if any of seven given formulas is strong enough to serve by itself as an axiom for the entire calculus. An automated theorem-proving program provided invaluable assistance in proving that five of the seven are too weak, where the proof rests on a total characterization of the theorems that can be deduced from each of the five. To obtain the characterizations, the program employed a new methodology based on the use of schemata [43]. The same program proved—contrary to conjecture—that the other two formulas are each strong enough to serve as a single axiom.

Before completing our discussion of open questions that were answered with the assistance of a theorem-proving program, let us pause to focus on the use of such a program to verify some given computer program. We take this detour because the final questions to be discussed are also relevant to programming languages, at least in a distant sense. To define program verification, we quote Boyer and Moore, from their contribution to the first issue of the Journal of Automated Reasoning [4]: “Computer programs may be regarded as formal mathematical objects whose properties are subject to mathematical proof. Program verification is the use of formal, mathematical techniques to debug software and software specifications.” (The article from which this quote is taken provides an excellent introduction to program verification and other applications of automated reasoning.)

Of the various efforts focusing on using an automated theorem-proving program for program verification, that of Boyer and Moore deserves the most acclaim. Their approach was to design and implement a program whose main purpose is to
verify algorithms and computer code, in contrast to programs designed to be
general-purpose. They refer to their program as a theorem prover, simply because
its use is based on proving one theorem after another. The theorems submitted to
the Boyer-Moore program [2] are obtained from the attempt to prove that some
given algorithm fulfills its intended purpose or from the attempt to prove that
some given computer program satisfies the claims made for it. As the following list
of successes shows, Boyer and Moore have indeed designed a powerful program, a
program that demonstrates that automated theorem-proving can be most useful for
program verification. Among the program's successes are proofs of

the invertibility of the Rivest-Shamir-Adleman public-key encryption algorithm
[3];

the soundness and completeness of a propositional-calculus decision procedure;

the soundness of an arithmetic simplifier;

the termination of Takeuchi's function;

the correctness of many elementary list and tree-processing functions;

the properties of many recursive functions; and

the verification conditions for various small Fortran programs, including a fast
string-searching algorithm, an integer square-root algorithm using Newton's
method, and a linear-time majority-vote algorithm.

Significantly larger programs have been verified with an automated theorem-
proving program. For example, the Stanford Verifier by Luckham and his students
at Stanford University [11] was used to verify a 3000-line Pascal compiler, and the
Gypsy Verification Environment (GVE) by Good [10] was used to verify a 4200-line
communications interface to a computer network. For the latter effort, some 2600
verification conditions were proved, using a theorem prover adapted from one by
Bledsoe [1].

We close this discussion on the application of automated theorem-proving to
program verification with one final citation. The success under consideration—
obtained by Smith, Wojcik, Chisholm, and Kljaich [6, 7, 12, 13]—concerns the
verification of software, hardware, and their interface in a system designed by
Draper Laboratories to replace certain failing sensors in a nuclear reactor at
Argonne National Laboratory. The reactor in question was designed in the later
1950s with the expectation of being used for approximately fifteen years, and
therefore the fact that its sensors are failing is no surprise. Since the reactor still
provides much useful data, the software/hardware replacement system merits
serious study. The automated theorem-proving program TTP [15, 16] played the key
role in the verification of the fault-tolerant (replacement) system designed by
Draper. In particular, TTP proved that the appropriate properties are present in the
Draper system; in addition, by examining the work performed by that reasoning
program, researchers found important simplifications that could be made to the
design of the system. The success of Smith and his colleagues is, in an obvious
sense, one of the best examples of the use of an automated reasoning program to
verify the claims made, at the highest level, for a system.
We now return to the topic of answering open questions with the assistance of
an automated theorem-proving program; our focus here is on combinatory logic.
This topic nicely combines aspects of pure mathematics and logic with the practical
objective of producing programs free of bugs. Researchers (for example, Turner
[30]) are most interested in compiling high-level applicative languages into combi-
nators—combinators that are the machine instructions executed by the computer
(see Sections 2.4 and 5 for examples of combinators and how they behave); in fact,
such computers have been designed and built. In our studies of combinatory logic,
we have been able to answer questions concerning the presence or absence, for
some given set of combinators, of what are called the weak and the strong
fixed-point properties (see Section 5 for their definitions). To appreciate the
difficulty of such questions, one might try to answer the first two challenge
questions we pose at the beginning of Section 5.

We were able to answer those questions and many other similar questions, and
the key is the use of our new global strategy, called the kernel strategy [37], for
systematically searching for fixed-point combinators. That success still startles us,
especially in that it marks a singular event for automated theorem proving. In
particular, there now exists an area (the study of the presence or absence of either
fixed-point property) in a deep field (combinatory logic) such that randomly
selected questions can be quickly answered with a high probability by submitting
them to a theorem-proving program. The kernel strategy has proved to be very
powerful, especially when applied by our recently designed automated theorem-
proving program OTTER [18]—which brings us naturally to the topic of the next
section, portable automated theorem-proving programs.

1.2.2. Portable Automated Theorem-Proving Programs. Each of the two fields,
automated theorem proving and logic programming, is empirical in nature. There-
fore, for the needed advances to occur, experimentation is essential. Indeed, the
likelihood and the frequency of significant advances are each directly dependent
on the amount of experimentation. As an illustration from a totally unrelated field,
one can show that the rapid progress made in the area of linear algebra resulted
directly from numerous experiments by various researchers with different com-
puter programs.

Unfortunately, the comparable and needed amount of experimentation did not
occur in the first twenty years of automated theorem proving. Although suspect,
one reason that was frequently given was the lack of access to a theorem-proving
program. The lack no longer exists. Indeed, we know of three automated
theorem-proving programs that are portable and in the public domain.

For program verification, one can use the Boyer-Moore theorem-proving pro-
gram [2]. The program, written in LISP, relies heavily on the use of induction and is
specifically designed to verify the correctness of computer programs and algo-
rithms. One can obtain the program by electronic mail. For the details, one should
write by electronic mail to boyer@cs.utexas.edu or by ordinary mail to the
Department of Computer Sciences of the University of Texas.

The second theorem-proving program, ITP [15, 16], is not tailored to any specific
application. This program, written in PASCAL, permits one to use it in interactive or
in batch mode. Use of ITP gives one access to a wide variety of inference rules and
strategies—some of which we shall discuss in Section 2—but this program does
not offer induction. VAX/UNIX, VAX/VMS, and IBM/CMS versions are available by contacting either of the following two sources:

Numerical Algorithms Group, Inc.
Suite 200
1400 Opus Place
Downers Grove, IL 60515-5702
Phone: (708) 971-2337
Fax: (708) 971-2706

National Energy Software Center
Argonne National Laboratory
Argonne, IL 60439
Phone: (708) 972-7172

The third portable automated theorem-proving program of interest is the very recently designed program OTTER [18]. OTTER, written in C, is a general-purpose theorem-proving program. Like ITP, it offers a wide range of inference rules and strategies, but does not offer induction. Among all of the theorem-proving programs of which we know, this new program runs faster than all but Aura [28]; Aura is faster at applying hyper-resolution, UR-resolution, and nonunit subsumption, and OTTER is faster at applying paramodulation, demodulation, and forward unit subsumption. Perhaps more significant—since experimentation of many types by many people is so valuable—OTTER can be used with a personal computer. Currently, for the details for obtaining a copy of OTTER, one should contact William McCune, by electronic mail at mccune@mcs.anl.gov, or by ordinary mail in the Mathematics and Computer Science Division at Argonne National Laboratory. However, very shortly, one will be able to obtain a copy of OTTER from either of the sources cited for ITP.

1.2.3. Researchers and Their Willingness to Experiment. The field of automated theorem proving and the field of logic programming are now beginning to emerge as significant areas of study and application, and that emergence can be traced directly to the new attitude toward experimentation. Specifically, researchers are far more willing now than ever before to test their ideas with actual problems and with existing computer programs. Further, when the appropriate computer program or routine is unavailable, researchers are more willing, and even eager, to design and implement what is needed than they were in the preceding many years.

In other words, until recently, the field of automated theorem proving suffered from an unwillingness to experiment. We are certain that this lack of a desire to test ideas with an existing program or—when such a program was not available—resistance to writing such a program was far less common in logic programming. Nevertheless, perhaps the research in logic programming was hampered by a related phenomenon—an unfamiliarity with what has been occurring in automated theorem proving. If that is the case, then this article may serve in part as an antidote.

Indeed, by being presented with new ideas or variations on old ideas, those interested in logic programming may—at least catalytically—be stimulated to enhance existing programs or, even better, design and implement even more
powerful and versatile ones than now exist. In particular, an examination of the material presented in Section 2, where we review the elements of automated theorem proving, may permit some researcher to adapt one or more of those elements to logic programming. Although we suspect that such an adaptation presents a formidable challenge, we cannot help but wonder what might result.

Of course, one of the key questions focuses on timing, on when to consider attempting to make the suggested adaptations. How astute F. Pereira was when he commented: "It is fortunate that the originators of logic programming had the courage to ditch equality in developing the concepts and techniques of logic programming... premature concern with generality would have prevented the development of the very efficient implementations that characterize current logic programming practice" [23]!

But Pereira was accurately discussing logic programming in its beginning, and we are focusing on 1989 and what is possible now. Therefore, especially in view of the high quality of current research in logic programming, perhaps the field is ready and able to employ more fully some of the features offered by the more powerful automated theorem-proving programs. In particular, as in its beginning, perhaps logic programming is ready to experiment heavily with techniques currently used in automated theorem proving: ready to experiment with incorporating the use of strategy far more powerful than currently used and with incorporating a more practical treatment of equality. The corresponding experiments would be most intriguing to us.

Because we prefer to see and perhaps use the results of those experiments sooner rather than later, the prudent course of action is to attempt to add to the current willingness and eagerness to experiment, and even attempt to increase the number of researchers involved in experimentation. In the remainder of this section, therefore, let us focus on the need for experimentation and also on some of the advances that have occurred directly because of experimentation.

With regard to the need for experimentation (recognizing that we are repeating ourselves, but also believing that this point deserves emphasis), most important of all, automated theorem proving and logic programming are each empirical sciences. Specifically, although advances in theory and advances in implementation play an equal and vital role—since the objective of both fields is that of diverse application—everything must be thoroughly tested, and tested by numerous users. The need for thorough testing is obvious. The need for numerous users rests with the fact that it is far too easy to develop a bias toward some approach.

For one of the less publicized examples of a bias that can be present, one can easily focus on questions, problems, or theorems from one area, not realizing that such a focus may distort one's evaluation. Equally—and also not discussed frequently—one can be inordinately convinced of the power of an approach by its success with simple problems or theorems. For an example of a simple theorem, one can study the theorem from group theory that asserts that the group is commutative whenever the square of every element $x$ is the identity. For an example of a theorem that is not simple or easy to prove, one might study the commutator theorem of group theory (see Section 2.6), which we pose as a challenge problem in Section 1.3. In contrast to testing an idea by a few individuals, if many individuals experiment with an idea—one of theory, implementation, or application—then the likelihood of a distortion occurring is sharply reduced.
With regard to advances in automated theorem proving that can be traced directly to experimentation, three that might be useful to logic programming immediately come to mind. Although we shall highlight these three in Section 2, let us quickly focus on each here. First, the \textit{set of support strategy} (see Section 2.3) was discovered because of experiments focusing on theorems from group theory, experiments designed and implemented to test the power of a program designed in the early 1960s by D. Carson of Argonne National Laboratory; the story of its discovery is given in Chapter 2 of [35]. As we shall learn, the object of the set of support strategy is to restrict the reasoning to information that is closely connected to the specific question under investigation. In a sense that we shall discuss in Section 2.3, logic programming employs a very restricted version of this strategy.

The second advance resulting from experimentation was the discovery of the procedure called \textit{demodulation} (see Section 2.4). That procedure was formulated to cope with the disappointing performance of a later version of the Argonne program when that program was asked to prove a simple lemma in number theory; see Chapter 2 of [35]. The object of demodulation is to rewrite information into a canonical form. We shall discuss in Section 2.2.3 the possible usefulness of this procedure to logic programming.

The third advance that can be traced to experimentation is the formulation of the \textit{weighting strategy} (see Section 2.3) [17]. This contribution of Overbeek resulted from his attempts to have his theorem-proving program find a proof of the commutator theorem. From our viewpoint, the object of the weighting strategy is mainly to enable the user to give a program hints about which information is significant and should therefore be keyed upon, and which is insignificant and should therefore be discarded. From Overbeek's viewpoint, the object of weighting is to circumscribe the terms that are conjectured to be relevant. We shall discuss in Section 4 the possible usefulness to logic programming of both the weighting and the set of support strategy.

With virtually no subtlety, we are suggesting that experimentation must be encouraged in as many ways as one can, and by as many people as one can interest. Fortunately—and here we again visit one of the factors that make us so optimistic—the current group of researchers is willing and, often, eager to experiment. Although the explanation for this dramatic change might rest with the simple fact that many people now conducting research have grown up with computers, we prefer to believe that the explanation rests with an understanding of the importance of experimentation.

1.2.4. Recent Advances. With the advent of parallel processing, the future looks even more enticing, and the opportunities are many. One prediction asserts that, by 1993, the Hypercube will offer 1024 processors, each of which will deliver 100 MIPS—and in parallel. Parallel-processing automated theorem-proving programs may be drawing 10,000 significant conclusions per second of CPU time; 10 MIPS may be required to draw a significant conclusion.

To fully grasp the importance of such an achievement, one might note that drawing a significant conclusion is equivalent to many LIPS (logical inferences per second). It would be incorrect to say that logic programming can already draw 10,000 conclusions a second, for LIPS should not be confused with conclusions per
second. Indeed, one "logical inference" for logic programming is somewhat equivalent to an attempt at one successful application of binary resolution, and it is difficult for a mathematician or logician to consider, for example, the concatenation of two lists as drawing many conclusions. Therefore, the term LIPS is a rather unfortunate choice, at least for mathematicians and logicians and the like.

With the sharp increase in the rate of conclusion drawing that results from the use of parallelism, with the advances that we expect in logic programming that also depend on parallel processing, and with the already existing evidence that automated theorem proving can borrow from logic programming (see Section 4), one can easily understand why access to this new type of computer produces so much optimism in us.

In addition to the increase in speed directly resulting from the use of parallel processing, even greater speed may be possible, depending on the results of certain research now in progress in logic programming—especially that under the aegis of the Gigalips project. We already have excellent examples of the value for automated theorem proving of some of the advances in logic programming. In the new program OTTER, the demodulation procedure (discussed in Section 2.4) for rewriting information into a canonical form employs techniques borrowed from logic programming. We shall give some of the pertinent details in Section 4 of this article. For a second example, an implementation of linked inference, which we shall introduce and discuss in some detail in Section 2.2.3, would also benefit from the use of certain techniques whose nature is reminiscent of logic programming.

For two other important advances, one might study the excellent research that focuses on compiling logic programs into efficient code, and also study the work of Overbeek and his colleagues [5] focusing on the use of PROLOG technology to implement a parallel-processing version of the inference rule hyperresolution. With regard to the former, in some unpublished experiments we have obtained impressive speed by compiling demodulators (see Section 2.4 for a discussion of that concept), an approach originating directly in the success in compiling logic-programming code. With regard to the latter, the full value of using logic programming technology for aspects of automated theorem proving must wait some years, but such use already appears to offer a great deal for an implementation of linked inference rules.

1.3. A Challenge and a Caveat

For those researchers who naturally question our enthusiasm concerning the various procedures on which a number of theorem-proving programs rest, we offer the following challenge problem. The challenge asks one to prove a theorem, called the commutator theorem (see Section 2.6 for more details), from abstract algebra, but, most important, to prove it strictly by means of logic programming. In particular, one is asked to seek a proof without using paramodulation, demodulation, subsumption, and weighting. The theorem to prove asserts that if the cube of every element $x$ in the group is the identity $e$, then $[[x, y], y] = e$ for all $x$ and $y$, where the commutator $[x, y]$ of any two elements $x$ and $y$ of a group is the product of $x$, $y$, the inverse of $x$, and the inverse of $y$.

Obtaining a proof of this theorem without using the key procedures discussed in Section 2 should provide a substantial challenge. Among the benefits that will
accrue to one who accepts the challenge are a deeper understanding and appreciation of why automated theorem proving requires the use of such complicated procedures as it relies upon and, perhaps, an impetus to attempt to adapt those same procedures for logic programming. To aid the reader in assessing progress in an attempt to meet the challenge, in Section 2.6 we supply two proofs of the commutator theorem.

Let us now turn to a review of automated theorem proving. For that review, although various paradigms exist for this field, we focus on that based on the use of clause language (see Section 2.1) and—in many respects—focus heavily on how we at Argonne treat automated theorem proving. We note that one of the authors (Wos) strongly favors the paradigm presented in Section 2 because the lack of richness of the clause language—which is considered a weakness by some—contributes, in his view, to a sharp increase in the probability of formulating effective inference rules and powerful strategies to control their application. We also note that, after discussing for a number of years the various approaches to automated theorem proving, it is Wos’s opinion that the paradigm of Section 2 is far more promising than the paradigms based on natural deduction, connection graphs, logic programming, and a nonclausal approach. Indeed, we conjecture that far more has been accomplished within the clause paradigm than within any other.

Therefore, as one reads Section 2, one might keep in mind that, although what we discuss applies to various paradigms for the automation of theorem proving and hence is not limited to our approach to automated theorem proving, the discussion is slanted in that direction. During that reading, one might also keep in mind the already discussed essential difference between automated theorem proving and logic programming; the former is oriented toward unalgorithmic reasoning and the latter toward algorithmic. Nevertheless, as we shall espouse—especially in Section 4—a natural symbiosis exists between the two fields.

2. A REVIEW OF AUTOMATED THEOREM PROVING

As one reads this rather extensive review of automated theorem proving, one may see many connections with logic programming; a more complete treatment of the larger field—automated reasoning—can be found in [38] or in [35]. Indeed, the basis for logic programming can be traced to Robinson’s formulation of the inference rule binary resolution [26] (see Section 2.2.1 below). Throughout the remainder of this article, we shall touch on other connections between logic programming and automated theorem proving, and pose questions concerning a possible closer connection that could exist between the two fields.

At the highest level, automated theorem proving—at least the paradigm that is our preference—can be viewed as imitating (in certain respects) mathematics by beginning its attack on a given question or problem by accruing various lemmas and conclusions. By retaining such information, automated theorem proving immediately departs from the usual practice in logic programming. The motive for information retention rests with the notion that, to succeed in answering deep questions and solving hard problems, the program cannot afford to continually rediscover the needed lemmas, facts, or relationships. The decision to retain information—as will become clear in this review—virtually requires the program
to rely on a number of procedures that might otherwise be unneeded, procedures for canonicalization, for information purging, and for restricting and directing its reasoning. However—as we shall suggest—avoiding the retention of information, and therefore bypassing the need for the additional procedures, sharply reduces the likelihood of success when the assignment concerns deep questions and hard problems.

Since unrestrained accrual of information is almost always doomed to failure, the program applies various types of strategy to restrict such accrual. In addition, unlike mathematics (since we as yet know of no way to avoid various types of redundancy), the theorem-proving program purges conclusions that are captured by more general information. Also unlike mathematics, the program applies types of reasoning that we would advise no person to apply: reasoning, such as paramodulation for generalizing equality substitution (see Section 2.2.2), that is tailored to the properties of a computer. Finally—as logic programming in effect does—in the vast majority of cases, the typical theorem-proving program seeks to prove the conjectured theorem by seeking a proof by contradiction.

If one quickly reviews the preceding remarks, one sees that, although similarities exist, automated theorem proving does indeed approach problem solving in a manner that is distinctly different in many respects from that found in logic programming. To gain some insight into why automated theorem proving—at least the paradigm that has worked for us—is so successful, to study the precise nature of each of its components, and to learn how strong our case for automated theorem proving is, let us review the basic elements.

We shall begin the review by briefly focusing on representation, specifically, on the clause language so familiar to researchers in logic programming. We must spend a little time on this topic because we observe conventions different from those of logic programming. We shall then turn to some of the types of reasoning offered by many theorem-proving programs, briefly touching on the inference rule binary resolution (already mentioned and familiar to many), heavily focusing on paramodulation and its use for “building in equality”, and also heavily focusing on linked inference rules and their use for avoiding many trivial intermediate deductions. We focus heavily on paramodulation because this inference rule, in our view, offers a most attractive approach for coping with equality and all of the attendant difficulties. We wonder whether there exists a way to use this inference rule to extend the power of logic programming without interfering with its impressive efficiency. We focus on linked inference rules because one of them, linked hyperresolution, provides—in terms that are used in automated theorem proving—a convenient way of discussing the reasoning applied in logic programming. A glance at linked inference rules might, at least catalytically, result in some interesting developments.

After completing our discussion of inference rules, we next turn to strategy of various kinds. Even more than the treatment of equality, in our view the area of automated theorem proving that offers substantial power for logic programming is that of strategy. We shall focus on strategies that restrict a program’s reasoning, and also on strategies that direct it. It is through the use of strategy that automated theorem proving derives much of its power.

The next topics in our review are demodulation, a procedure for rewriting information into a canonical form, and subsumption, a procedure for discarding a
particular type of logically weak information. Just as the use of strategy is crucial, a
program that attempts to answer deep questions or solve hard problems is almost
always doomed to failure without the use of demodulation. Subsumption, whose
use is also vital, might at first appear to hold little or no interest for the logic
programming community. Nevertheless, a discussion of this procedure and its
relation to theorem-proving programs implemented in the spirit of logic program-
ming may—at least for some—prove exceedingly provocative.

To provide at least a taste of all of the key elements of automated theorem
proving, we close our review with a discussion of proof by contradiction. This topic
is in an obvious sense very familiar to the experienced researcher in logic
programming. Nevertheless, certain examples of proof by contradiction might be of
some interest, especially those focusing on the commutator theorem as an example
of a rather difficult theorem to prove. Indeed, that theorem exemplifies the various
points we make in this section, and provides the type of challenge (Section 1.3)
that might prove stimulating, for consideration of the theorem shows why many of
the procedures we discuss are needed. To prepare for our treatment of each of the
specific elements of automated theorem proving, in which we rely mainly on
examples rather than on formal definitions, let us immediately summarize the
actions taken by a typical theorem-proving program.

The fields of mathematics and logic from which a theorem may be taken include
algebra, geometry, topology, set theory, and combinatory logic. The most widely
used approach for proving theorems—or for solving any problem or answering any
question—with a theorem-proving program focuses on searching for a proof by
contradiction. Therefore, to enable a program using such an approach to attempt
to prove a proposed theorem of the form if $P$ then $Q$, one is required to define the
field from which the theorem is taken, to give the special hypothesis (which may be
the empty set) of the theorem, and to assume that the theorem is in fact false. By
the special hypothesis, we mean that which remains of the if conditions of a
theorem when one ignores the axioms of the theory from which the theorem is
taken and also ignores any lemmas that the user supplies. We also use the term
denial of the conclusion to refer to those statements—or clauses—that arise from
assuming the theorem false. As we discuss in Section 2.3, the special hypothesis
and the denial of the conclusion each play an important role in the use of a
powerful strategy, the set of support strategy.

To begin its attack on the given problem, the theorem-proving program draws
conclusions—some of which it retains—by applying one or more sound inference
rules chosen by the researcher. The retention of conclusions markedly increases
the effectiveness of a theorem-proving program and also permits one to use such a
program to explore a new theory, and then display some of the properties of that
theory. The application of each rule is restricted by one type of strategy and
directed by another type. When a conclusion is drawn, the program rewrites it into
a canonical form by applying rules (demodulators [41]) supplied by the researcher
or discovered by the program. The result is then tested (with the use of subsump-
tion [26]) to see whether it is a trivial corollary of information the program already
has, and should therefore be discarded. The result can also be tested—by using
weighting [17] (see Section 2.3 below)—to see whether it satisfies criteria (weights),
supplied by the researcher, for measuring significance; if the conclusion fails the
test, it is discarded. As expected, the goal is to deduce some new conclusion that
contradicts a conclusion drawn earlier or contradicts some input statement. When such a deduction is made, a proof by contradiction has been found, and the automated theorem-proving program in use "knows" that the given assignment has been completed.

Using the preceding summary of the program's actions as an outline for this section, let us sample each of the main areas, beginning with representation—a discussion of how one specifies the problem to be considered. Of course, for the experienced researcher in logic programming, this first topic will be in the main a familiar one.

2.1. Representation

In this section, we mainly use the notation and conventions employed in the two books we recommend for further study [38, 35], a notation that is consistent with the formal language—the clause language—most commonly used in automated theorem proving and so familiar to logic programming. We include some discussion because our notation and conventions differ from those frequently used in logic programming.

Our conventions require variables to be chosen from among terms whose first symbol is one of lowercase u through z (which contrasts with the uppercase notation used in various dialects of logic programming). Each variable is implicitly universally quantified (meaning "for all"), and each variable is relevant only to the clause in which it occurs. Also by convention, functions and constants are (almost always) written in lowercase, and relations (predicates) written in uppercase. In addition—and this convention is most important—by writing \texttt{EQUAL} for the equality relation, our programs are able to treat equality as "understood" or "built in". Finally, the symbol \( \neg \) means \texttt{not}, and the symbol \( \lor \) means \texttt{or}. In other words, one encounters another notational difference from logic programming, for we explicitly use a symbol for logical \texttt{or} and for logical \texttt{not}. As in logic programming, existence is not represented with variables; instead, appropriate functions and constants are employed. In contrast with logic programming, the logical operator \texttt{and} never occurs within a clause.

Some of the differences between the standard PROLOG notation and that we use now in automated theorem proving are captured by the following example:

\begin{verbatim}
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
GRANDPARENT(X, Y) :- \neg PARENT(X, Z), \neg PARENT(Z, Y).
\end{verbatim}

The first of the two given clauses is in PROLOG notation, and the second is in the notation we (usually) employ in automated theorem proving.

Of course, as is well known, one of the key differences between typical logic programming and automated theorem proving is the latter's use of clauses that contain as many positive literals as desired. By not being restricted to the use of Horn clauses (clauses containing at most one positive literal), one can rely on a more natural representation. In our next example of the notation and conventions we use, we shall encounter clauses that are not Horn. It may well be—even though we would like to believe otherwise—that the successes of logic programming are
so dependent on the restriction to the use of Horn clauses that all other options are unacceptable in the vast majority of cases.

For the final difference between logic programming and automated theorem proving, we note that (in the latter) the denial of a theorem might be represented with clauses each of which contains positive literals only. Obviously (for logic programming) such a clause cannot be used as a goal clause, which can force one to use an axiom, for example, as the goal clause. As we comment later, basing a search for information entirely on an axiom can be disastrous. Let us now turn to more interesting examples.

To illustrate our notation, to provide an example that might be challenging to consider for logic programming, and to exemplify (in Section 2.2) various inference rules, let us consider the following theorem from abstract group theory: If a subgroup $O$ has index 2 in a group $G$, then $O$ is a normal subgroup. (For the axioms of a group—first in a notation that avoids in the main the use of equality and second in a notation that emphasizes equality—one can turn to the commutator theorem in Section 2.6.) By definition, a subgroup $O$ has index 2 in the group $G$ if and only if $G/O$ ($G$ mod $O$) consists of two left cosets, the elements in $O$ and the elements not in $O$. Also by definition, a subgroup $O$ is a normal subgroup if and only if the product of $x$, $y$, and the inverse of $x$ is an element of $O$ for all $x$ in the group and all $y$ in $O$. Where equality is axiomatized rather than built in, the following clauses can be used—if one also uses clauses (GT-1) through (GT-17) of Section 2.6—to present the theorem under discussion to an automated theorem-proving program:

$O$ is a subgroup:

(SIT-1) $O(e)$

(SIT-2) $\neg O(x) \lor O(inv(x))$

(SIT-3) $\neg O(x) \lor O(y) \lor P(x,y,z) \lor O(z)$

$O$ has index 2:

(SIT-4) $O(x) \lor O(y) \lor P(x,i(x,y),y)$

(SIT-5) $O(x) \lor O(y) \lor O(i(x,y))$

Denial of the conclusion:

(SIT-6) $P(b,inv(a),c)$

(SIT-7) $P(a,c,d)$

(SIT-8) $O(b)$

(SIT-9) $\neg O(d)$

For a proof of this theorem in clause notation, see Section 6.1.1 of [35], which completes the representational discussion of the index-2 theorem.

To show how equality can be treated to enable the program to “understand” that concept, let us turn to examples from ordinary arithmetic—or, alternatively, from group theory written additively. To express the fact that there exists a left identity,

$0 + x = x$

for all $x$, we write

`EQUAL(sum(0,x),x)`

as the corresponding clause. To express the existence of an additive (right) inverse,
for all \( x \) there exists a \( y \) such that
\[
x + y = 0,
\]
we write
\[
\text{EQUAL}(\text{sum}(x,\text{minus}(x)), 0)
\]
as the clause.

Having dispensed with representation, let us now turn to the types of reasoning that can be offered by an automated theorem-proving program.

2.2. Inference Rules

In this subsection, we focus on some of the inference rules that the more versatile automated theorem-proving programs offer. We begin with binary resolution [26] (because researchers in logic programming are so familiar with it, and because it plays such an important role for logic programming), UR-resolution [17], and hyperresolution [27]. We next turn to paramodulation [25, 39], the inference rule that can be used to enable a theorem-proving program to treat equality as if it were “understood”. We close this section by focusing on two linked inference rules: linked UR-resolution and linked hyperresolution [42].

To provide an overall comparison of the inference rules discussed here, we note that—in contrast to binary resolution—UR-resolution and hyperresolution can each consider simultaneously two or more clauses at a time, provided that certain conditions are met. The use of UR-resolution emphasizes the role of unit clauses (clauses with one literal), and the use of hyperresolution emphasizes the role of positive clauses (clauses without negative literals). Paramodulation—which is the best example of a computer-oriented inference rule, and one that a person probably should not apply by hand—generalizes the usual notion of equality substitution (see the third example in Section 2.2.2) and requires the use of the clause
\[
\text{EQUAL}(x, x)
\]
for reflexivity, but requires no other clauses for that relation.

Because of the importance of coping with equality, among the inference rules we discuss here, paramodulation may be of greatest interest to logic programming. Indeed, to attempt to solve hard problems from mathematics, a program is often required to reason within deeply nested expressions, which paramodulation does, and does without being forced to process all of the intermediate expressions that must be processed when equality is treated with the usual set of axioms. Linked UR-resolution and linked hyperresolution—the latter of which will be seen to capture the type of reasoning used in logic programming—generalize the corresponding inference rule, and are designed to avoid deducing certain types of intermediate and trivial information.

2.2.1. Binary Resolution, UR-Resolution, and Hyperresolution. Because an understanding of the inference rule paramodulation does not come easily, and because we consider a built-in treatment of equality to be crucial for automated theorem proving, let us begin by focusing on simpler inference rules. As promised,
we can use the theorem that asserts that subgroups of index 2 are normal. For our first three examples, we can use the following clauses:

(SIT-2) \(- O(x) \mid O(\text{inv}(x))\)
(SIT-3) \(- O(x) \mid \neg O(y) \mid \neg P(x,y,z) \mid O(z)\)
(SIT-4) \(O(x) \mid O(y) \mid P(x,i(x,y),y)\)
(SIT-8) \(O(b)\)
(SIT-9) \(- O(d)\)
(GT-7) \(P(x,y,\text{prod}(x,y))\)

The last clause is sometimes called the \textit{closure axiom}, and asserts the \textit{totality} of the function \text{prod}.

As is well known to the people in logic programming, at each step on each path designed to solve a subgoal, the reasoning employed by logic programming is in effect an attempt to successfully apply a single binary-resolution step. For example, by applying binary resolution to clauses (SIT-8) and (SIT-2), we obtain

\[ O(\text{inv}(b)) \]

as the conclusion, which would also be obtained by applying UR-resolution or hyper-resolution to this pair of clauses.

For a more interesting example of the use of UR-resolution—one that captures the spirit of the rule—we simultaneously consider clause (SIT-4) and two copies of clause (SIT-9) to obtain

\[ P(d,i(d,d),d) \]

as the conclusion. The inference rule UR-resolution considers a nonunit clause (the nucleus); with the exception of one literal, it attempts to simultaneously unify each of its literals with a unit clause (a satellite) and, if successful, produces a unit clause as the conclusion. The use of UR-resolution, therefore, permits the program to deduce a negative unit clause as well as a positive unit clause, and—in contrast to hyper-resolution—to focus on a set of hypotheses involving more than one clause containing negative literals.

The inference rule hyper-resolution considers a nonpositive clause (the nucleus); it attempts to simultaneously unify each of its negative literals with a literal in positive clause (a satellite) and, if successful, produces a positive clause (possibly the empty clause) as the conclusion. For hyper-resolution to obtain the conclusion \(P(d,i(d,d),d)\) from the same hypotheses, two steps would be required. In particular—although not in the spirit of hyper-resolution, which encourages the deduction of clauses some of whose positive literals come from the clause containing the negative literals to be removed—the program could first apply the inference rule to clauses (SIT-9) and (SIT-4) to obtain

\[ O(y) \mid P(d,i(d,y),y), \]

which, with a second application of hyper-resolution, would then be used with clause (SIT-9) to again reach the conclusion for which UR-resolution required but one deduction step. For an application of hyper-resolution that is more typical, we apply the rule to the simultaneous consideration of clauses (SIT-3), (SIT-8),
THEOREM PROVING AND LOGIC PROGRAMMING

(SIT-8), and (GT-7) to obtain

\[ O(\text{prod}(b, b)) \]

as the conclusion. We note that neither UR-resolution nor hyperresolution, as the examples show, is restricted to the consideration of Horn clauses.

Unless purged with some procedure such as subsumption (see Section 2.5) or weighting (see Section 2.3), the cited conclusions would be retained. Retention of clauses—as the reader no doubt realizes—is motivated by the desire to use them later in the attempt to complete the given assignment, and by the desire to save the CPU time, which might be as much as 15 minutes, that would be needed to deduce them again. A less obvious motivation for the retention of clauses rests with the side effect of avoiding numerous undesirable paths, which we discuss in Section 2.2.3. The tricky part, as is obvious, is to decide which information to retain and which to purge, or treat as implicitly present—an additional topic we address in Section 2.2.3 when we discuss linked inference rules. Just in case one is skeptical about the need to retain any information versus the desire to save CPU time, we suggest that one examine the commutator theorem from group theory and its two proofs, which we give in Section 2.6.

Of course, to save all conclusions is madness. Indeed, one motivation for using hyperresolution or UR-resolution or some linked inference rule is to sharply reduce the number of conclusions to be retained. Although hyperresolution, for example, can be viewed and programmed as a sequence of binary-resolution steps, it can also be programmed—as Overbeek did in AURA [21]—without generating any intermediate clauses. In the same sense, linked inference rules can be viewed and programmed as a sequence of binary resolution steps, but we are almost certain that a far better choice is to borrow from logic programming to implement such rules. In other words, we have an excellent example of how logic programming can be of use to automated theorem proving. For the converse, let us now turn to a discussion of an inference rule—paramodulation—whose use directly addresses the difficulties presented by equality.

2.2.2. Paramodulation, Defined with Examples. We now come to one of the two areas that we conjecture offers the most for logic programming, one being our treatment of the equality relation (see Sections 2.2.2 and 2.4), the other being strategy (see Section 2.3). There now appears to be ample evidence that treating equality as a built-in concept is far superior to an axiomatic approach. In particular, by using paramodulation—which we feature in this section—an automated theorem-proving program can reason directly within deeply nested expressions rather than touching the various so-called intermediate points. For a trivial example, paramodulation can use the fact that \( a = b \) to immediately deduce in one step that

\[ Q(f(g(h(b)))) \]

from the fact that

\[ Q(f(g(h(a)))) \]

without deducing the intermediate conclusions that would be deduced by applying the appropriate axioms for equality. The key is that paramodulation is a term-ori-
mented inference rule rather than a literal-oriented rule. This rule is definitely computer-oriented rather than person-oriented. Although the need for strategies to control the application of this inference rule is great, its use often yields impressive results, as we comment on in Section 2.6 when we discuss the commutator theorem of group theory—a proof of which was obtained in less than 3 CPU seconds by using paramodulation.

Rather than giving a formal definition of paramodulation, we shall, as in the preceding section, rely on examples. The first example illustrates the use of the simplest and most familiar form of equality substitution. The second illustrates a slightly more complex form of substitution—a form that will immediately seem reasonable—in that variables occur in one of the hypotheses, in the into clause. The third example, substantially more complex, is by far the most interesting, for it shows how paramodulation generalizes the standard notion of equality substitution.

For the first example, paramodulation applied to both the equation \( a + (-a) = 0 \) and the statement that \( a + (-a) \) is congruent to \( b \) yields in a single step the conclusion or statement that \( 0 \) is congruent to \( b \). In clause form, from

\[
\text{EQUAL}(\text{SUM}(a, \text{MINUS}(a)), 0)
\]

into

\[
\text{CONGRUENT}(\text{SUM}(a, \text{MINUS}(a)), b)
\]

one obtains the clause

\[
\text{CONGRUENT}(0, b)
\]

by paramodulation.

For the second example, paramodulation applied to both the equation \( a + (-a) = 0 \) and the statement that \( x + (-a) \) is congruent to \( x \) yields in a single step the conclusion or statement that \( 0 \) is congruent to \( a \). In clause form, from

\[
\text{EQUAL}(\text{SUM}(a, \text{MINUS}(a)), 0)
\]

into

\[
\text{CONGRUENT}(\text{SUM}(x, \text{MINUS}(a)), x)
\]

the clause

\[
\text{CONGRUENT}(0, a)
\]

is obtained. This example illustrates some of the complexity of paramodulation; in particular, the second occurrence of the variable \( x \) in the into clause becomes the constant \( a \) in the conclusion, but the term containing the first occurrence of \( x \) becomes the constant \( 0 \) in the conclusion. Although the unification of the first argument of the from clause with the first argument of the into clause temporarily requires both occurrences of the variable \( x \) to be replaced by the constant \( a \), paramodulation then requires an additional term replacement justified by equality substitution, the replacement of \( a + (-a) \) by \( 0 \).

Finally, for the third and complex example, paramodulation applied to both the equation \( x + (-x) = 0 \) and the equation \( y + (-y + z) = z \) yields in a single step the conclusion \( y + 0 = -(-y) \). In clause form, from

\[
\text{EQUAL}(\text{SUM}(x, \text{MINUS}(x)), 0)
\]
THEOREM PROVING AND LOGIC PROGRAMMING

\[ \text{EQUAL}(\text{sum}(y, \text{sum}(-y, z)), z) \]

the clause

\[ \text{EQUAL}(\text{sum}(y, 0), -\text{sum}(-y)) \]

is obtained by paramodulation.

To see that this last clause is in fact a logical consequence of its two parents, one unifies the argument \( \text{sum}(x, -\text{sum}(x)) \) with the term \( \text{sum}(\text{sum}(y), z) \), applies the corresponding substitution to both the \textit{from} and \textit{into clauses}, and then makes the appropriate term replacement justified by the typical use of equality. The substitution found by the attempt to unify the given argument and given term requires substituting \( -y \) for \( x \) and \( -\text{sum}(-y) \) for \( z \). To prepare for the (standard) use of equality in this third example—and here we encounter a key feature of paramodulation—a nontrivial substitution for variables in both the \textit{from} and the \textit{into clauses} is required, which illustrates how paramodulation generalizes the usual notation of equality substitution. In contrast, in the standard use of equality substitution, one does \textit{not} apply a nontrivial replacement for variables in both the \textit{from} and the \textit{into} statements.

Summarizing, a successful use of paramodulation combines in a single step the process of finding the (in an obvious sense) most general common domain for which the \textit{from} and \textit{into clauses} are both relevant and applying standard equality substitution to that common domain. The complexity of this inference rule rests in part with (1) its unnaturalness (if viewed from the type of reasoning people employ), (2) the fact that the rule is permitted to apply nontrivial variable replacement to both of the statements under consideration, and (3) the fact that different occurrences of the same expression can be transformed differently. As an illustration of the third factor contributing to the complexity of paramodulation, in the last example we gave of its use, the (term containing the) first occurrence of \( z \) is transformed to 0, but the second occurrence of \( z \) is transformed to \( -\text{sum}(-y) \). The third example illustrates what we mean when we say that paramodulation is a term-oriented inference rule that generalizes equality substitution.

A single application of paramodulation can lead to the deduction of an interesting conclusion that the usual mathematical reasoning requires more than one step to yield. In contrast to paramodulation, the mathematician or logician picks the instances—the substitutions of terms for variables—that might be of interest, and then applies (the usual notion of) equality substitution. Paramodulation, on the other hand, picks the (maximal) instances of the \textit{from} and \textit{into clauses} automatically and—rather than deducing intermediate conclusions by applying the corresponding replacement of terms for variables—combines the instance picking with a standard application of equality substitution.

As with logic programming, avoiding the retention of intermediate conclusions can contribute markedly to the performance of a theorem-proving program. The instances that a person chooses may be far less general than those chosen by the theorem-proving program in its attempt to apply paramodulation. As a result, the program may deduce a conclusion that is far more general than a person might deduce. Again the program benefits, for its efficiency often increases by having
access to general conclusions rather than specific ones; people do not require such
generality in most cases.

Summarizing, the inference rule instantiation—replacing variables by terms in a
statement to deduce a corollary—is frequently used, and used wisely, in mathemat-
ics and logic. For example, a mathematician might use instantiation to deduce that
\((yz^r yz) = e\) (the identity in a group) from the hypothesis that \((xx) = e\), and then
use the new conclusion as a key step in proving that the type of group under study
is commutative. A typical theorem-proving program would not draw such a
conclusion, since instantiation is ordinarily not an inference rule offered by such a
program, for no known strategy exists for wisely choosing appropriate instances
even though mathematicians and logicians do have that ability.

To give a second illustration of the power offered by paramodulation—if
combined with a procedure (demodulation, see Section 2.4) used by theorem-
proving programs to simplify expressions—we consider the following example, so
typical of mathematics. In the example, the conclusion that is reached with a string
of equalities (if one applies the usual type of reasoning that occurs in mathematics)
is drawn by a theorem-proving program in a single deduction step:

\[
z = 0 + z = (x + (-x)) + z = x + ((-x) + z).
\]

This string of equalities constitutes a proof of an obvious but useful lemma in
arithmetic and in group theory, the lemma that asserts the equality of the first and
last expressions in the string of equalities. In clause notation, from right inverse

\[
\text{EQUAL}(\text{sum}(x, \text{minus}(x)), 0)
\]

into associativity

\[
\text{EQUAL}(\text{sum}(\text{sum}(u, v), z), \text{sum}(u, \text{sum}(v, z)))
\]

the program can deduce

\[
\text{EQUAL}(\text{sum}(0, z), \text{sum}(x, \text{sum}(\text{minus}(x), z)))
\]

which it can then automatically simplify (see Section 2.4) to

\[
\text{EQUAL}(z, \text{sum}(x, \text{sum}(\text{minus}(x), z)))
\]

without producing any intermediate results.

To gain some insight into the power of paramodulation and to experience its
usefulness, one can compare the two proofs of the commutator theorem that we
give in Section 2.6. Even further, one might attempt to obtain a proof of this
theorem with whatever program is available. If that program is based strictly on
logic programming, then (especially since the sets of clauses for each proof consist
of Horn clauses) such an attempt may yield important information.

2.2.3. Linked Inference Rules. At this point, in contrast to focusing on a single
inference rule, we turn to a class of inference rule—linked inference rules \([42]\). As
we discuss linked inference rules, those who are familiar with logic programming
may experience pleasure at recognizing certain properties shared by these infer-
ence rules and the type of reasoning employed in logic programming. It may also
be interesting to note that part of the motivation for the formulation of these rules
is clearly reminiscent of one of the forces that caused logic programming to come
into existence—the desire to curtail information retention. Let us begin with a touch of history.

From approximately twenty years of experimentation, we concluded—not with a rigorous proof—that one important source of ineffectiveness for automated theorem-proving programs is the retention of far too much information, especially the intermediate information that is used mainly to bridge the gaps between more interesting bits of information. The notion was that bridging clauses are in an important sense less significant and should therefore be prevented from playing an equal and active role in the program's reasoning; in fact, such clauses should be present implicitly only.

To achieve this end, we chose to switch the focus of attention from the bridging (less significant) clauses to certain of their parents. For example (taken from the "jobs puzzle" discussed later in this section), rather than considering the bridging clause

\[-\text{MALE}(\text{Roberta})\]

as the focus of attention, we switched the focus to the parent clause

\[-\text{FEMALE}(\text{x}) | -\text{MALE}(\text{x})\]

from which the literal in the bridging clause is descended. By switching the focus of attention, we could force bridging clauses—which are not necessarily unit clauses—to be present implicitly only. We began to think of parent clauses of the type just given as links or linking clauses (not to be confused with the term links as used in connection graphs). The objective was to restrict the role of linking clauses, requiring them to play a subordinate role in any deduction step in which they are present by using them only to connect other pairs of clauses. To impose the desired restriction, we formulated the class of inference rule, linked inference rules, on which we focus in this section. Although somewhat hidden, part of the objective in formulating these new rules was the desire to generalize familiar inference rules and, even more important, the desire to permit a theorem-proving program to retain only significant bits of information in contrast to avoiding information retention entirely.

Simply stated, the need to retain some new information rests with the desire to use an automated theorem-proving program to attack deep questions and hard problems, many of which are taken from mathematics. In mathematics, one often needs to repeatedly apply various lemmas to prove the theorem under study, and some of those lemmas require many CPU minutes to prove. In such a case, one cannot afford to repeatedly prove the needed lemmas. Even more important, if one retains no new information, then there exists the potential for pursuing a huge number of paths in search of a proof. To sharply reduce that number—and perhaps to sharply reduce the amount of backtracking—a program can employ demodulation (Section 2.4) and subsumption (Section 2.5).

To see how demodulation is useful, how its use sharply reduces a search for information, and how its use increases the effectiveness of a search for a proof, let us imagine that we are studying group theory. Among the demodulators (rules for rewriting information into a canonical form) that we recommend for inclusion in the input for such a study is the axiom for left identity

\[\text{EQUAL}(\text{prod}(e, x), x)\]
with \( e \) denoting the identity of a group. If clauses are retained and demodulation is used, then the program will never consider a path involving the clause

\[
\text{ELEMENT}(\text{prod}(e,a),h)
\]

(the product of \( e \) and \( a \) is an element of the set \( H \)) once it has considered the clause

\[
\text{ELEMENT}(a,h),
\]

because the first of the two clauses in the predicate \text{ELEMENT} will be demodulated to the second, and then discarded by using subsumption; the two clauses correspond to versions of essentially the same fact. Even more annoying than pursuit of the type of path just illustrated is the case in which the program discovers that \( a = b \), but then either fails to demodulate one to the other because no equivalent of demodulation is in use, or cannot demodulate one to the other, as occurs when clauses are not retained. In contrast to what we conjecture is crucial for attacking problems from mathematics and logic—namely, the retention of clauses—without demodulation, various versions of essentially the same fact will be available to cause the program to pursue corresponding paths.

With regard to subsumption by itself, if the program deduces

\[
\text{EQUAL}(a,a)
\]

as a fascinating conclusion, the use of subsumption will immediately cause the discarding of that clause in the presence of

\[
\text{EQUAL}(x,x)
\]

and thus avoid various paths of inquiry. This last clause—reflexivity of equality—is always present when equality is relevant, even when paramodulation is used. Without retaining clauses and using demodulation and subsumption, we thus see how a program can pursue numerous paths that would otherwise be avoided.

In contrast to the situation in which one wishes a program to assist in answering some deep question or some hard problem that requires a general search for information, we know of two cases in which the choice to avoid information retention seems very wise. The first case focuses on reasoning that is algorithmic in character. When one knows the precise paths to pursue to reach the desired answer or to obtain the desired proof, then, indeed, why retain any intermediate information—why not consider all clauses that are in the input but are not nuclei or satellites as links to the desired result? (The terms nucleus and satellite are informally defined in Section 2.2.1, and examples relevant to linked UR-resolution and to linked hyperresolution are given in the discussion of the "jobs puzzle" found later in this section.) We in fact applaud logic programming for taking this approach when presented with problems for which one can find an appropriate algorithm to apply.

The second case concerns the proof of extremely simple theorems. For such theorems, one can profitably apply a simple strategy (see Section 2.3) and can afford to pursue a few distinct paths before completing a proof; backtracking can be kept to a minimum. After all, if the theorem under study is easy to prove, then the program's chance of succeeding may indeed be high and may require the traversal of only short paths, and not very many of those either.
Unfortunately, one seldom knows before the fact whether a theorem is easy to prove. Therefore—and let us be explicit in this regard—we strongly favor the approach that retains a number of clauses when the assignment concerns the proof of some theorem: we conjecture that automated theorem-proving programs that avoid essentially all retention of new information will prove to offer insufficient power. We shall say more about this issue in Section 5. However, since the needed experimental data for validating or refuting our conjecture are still lacking, we are pleased to note that researchers are studying the approach to theorem proving that is based on the spirit of logic programming.

Given our view concerning the need to retain information—and noting that, although we have found standard inference rules like those discussed in Section 2.2.1 to be useful, their use often causes a program to retain too many clauses—the question is what to do next. One obvious answer is to generalize, and one clue as to how to make appropriate generalizations rests with the desire to give an automated theorem-proving program the capacity to “jump to conclusions”, provided that the conclusions follow logically. For example, if in some puzzle one knows that the job of nurse is held by a male and that Roberta has the job of teacher or nurse, one jumps correctly to the conclusion that Roberta is not a nurse. Of course—in addition to the missing fact that Roberta must be female—the key missing fact asserts that people who are female are not male. In particular, the clause

\[
\neg \text{FEMALE}(x) \quad \text{or} \quad \neg \text{MALE}(x)
\]

links the clause

\[
\text{FEMALE}(\text{Roberta})
\]

to the clause

\[
\neg \text{HASJOB}(x,\text{Nurse}) \quad \text{or} \quad \text{MALE}(x)
\]

to draw the correct conclusion that Roberta is not the nurse.

We say “jumps to the conclusion” because a person ordinarily does not explicitly conclude—or even notice that an intermediate conclusion has been drawn—that Roberta is not male. We surmised (with no stroke of brilliance) that if an automated theorem-proving program could reason similarly, then the program would be more effective in its reasoning. Therefore, we formulated linked inference rules accordingly: rules that would treat certain bits of trivial information as present implicitly only, rather than having such information explicitly present in the database of facts and relations.

For our first illustration—to remove any distractions resulting from a lack of familiarity with the problem domain—we focus on a fragment from the “jobs puzzle” (see Chapter 3 of [38]), delaying the presentation of a far more interesting example of deductive reasoning until Section 2.6. The jobs puzzle concerns four people (Roberta, Thelma, Steve, and Pete) and eight jobs (actor, boxer, chef, guard, nurse, police officer, teacher, and telephone operator). Each person holds two jobs. You are asked to determine which jobs are held by which people. Among the clues, you are told that Roberta is not the chef, and that the husband of the chef is the telephone operator.
One immediately and correctly deduces—jumps to the conclusion—that Thelma must be the chef. A brief examination of the following five clauses show that UR-resolution can be used to reach the same conclusion, but three steps are required rather than the one that appears to be sufficient. On the other hand, as we shall show, by applying linked UR-resolution, one step is in fact enough. In addition, where standard UR-resolution (with the given clauses) requires the program to key on a general fact (J-5) rather than on the specific fact (J-1) that should initiate the deduction, linked UR-resolution behaves as one prefers by using the better choice to initiate the deduction as one would hope—by keying on clause (J-1).

Let us consider the following five clauses, and then focus on standard UR-resolution versus linked UR-resolution:

Roberta is not the chef.
\((J-1) \sim \text{HASAJOB} (\text{Roberta, chef})\)

If a job is held by a female, then the female is Roberta or Thelma.
\((J-2) \sim \text{FEMALE} (\text{jobholder}(y)) \rightarrow \text{HASAJOB} (\text{Roberta, } y) \lor \text{HASAJOB} (\text{Thelma, } y)\)

If a person is a wife, then that person is female.
\((J-3) \sim \text{HUSBAND} (x, y) \rightarrow \text{FEMALE} (y)\)

The person who holds the job of telephone operator is the husband of the person who holds the job of chef.
\((J-4) \sim \text{HASAJOB} (x, \text{telop}) \lor \text{HUSBAND} (x, \text{jobholder} (\text{chef}))\)

For every job, there is a person holding that job.
\((J-5) \exists \text{jobholder} (\text{job}(y))\)

UR-resolution suffices to yield the desired conclusion (J-8) in three steps. From clause (J-5) as satellite and clause (J-4) as nucleus,
\((J-6) \text{HUSBAND} (\text{jobholder} (\text{telop}), \text{jobholder} (\text{chef}))\)
is obtained. From clause (J-6) as satellite and clause (J-3) as nucleus,
\((J-7) \text{FEMALE} (\text{jobholder} (\text{chef}))\)
is obtained. And finally, from clauses (J-7) and (J-1) as satellites with clause (J-2) as nucleus,
\((J-8) \text{HASAJOB} (\text{Thelma, chef})\)
is deduced as the desired result.

A person solving this puzzle fragment would simply and naturally conclude clause (J-8) by (simultaneously) considering the information contained in clauses (J-1) through (J-5). The information contained in clauses (J-6) and (J-7) would not exist, at least explicitly. One variant of linked UR-resolution would also immediately deduce clause (J-8), but—in contrast to standard UR-resolution—by simultaneously considering clauses (J-1) through (J-5) and without deducing clauses (J-6) and (J-7). In particular, in the terminology sometimes used in discussing linked inference [42], the user could instruct a reasoning program to choose clause (J-1) as the initiating satellite and the predicate \text{HASAJOB} as the target. With those choices, clause (J-2) is the nucleus, and clauses (J-3) and (J-4) are linking clauses that \text{link} clause (J-2) to the satellite, clause (J-5). Clauses (J-6) and (J-7) are present implicitly only.
For a simple example of linked hyperresolution—and, because the clause on
which we focus consists solely of negative literals, one that reflects some of the
spirit of logic programming—we can again borrow from the jobs puzzle, but with a
slight twist. We consider clauses (J-9) through (J-12):

The job of nurse is held by a male
(J-9) \neg \text{HASJOB}(x, \text{nurse}) \lor \text{MALE}(x)

Everyone is not female or not male
(J-10) \neg \text{FEMALE}(x) \lor \neg \text{MALE}(x)

Roberta is female
(J-11) \text{FEMALE}(	ext{Roberta})

Roberta has the job of nurse
(J-12) \text{HASJOB}(	ext{Roberta}, \text{nurse})

letting clause (J-10) be the nucleus—the clause whose negative literals are to be
removed—or, using the terminology of logic programming, the goal clause. The
object (which is the slight twist) is to prove that Roberta is not the nurse, which
explains the presence of clause (J-12). With linked hyperresolution, clause (J-10)
could be used to initiate the search, and thus play the role of an initiating nucleus.
Alternatively, clause (J-11) could be used as an initiating satellite to unify with one
of the literals of clause (J-10), and clause (J-9) could be used both to unify with the
other literal of clause (J-10) and to link clause (J-10) to clause (J-12).

This example illustrates how linked hyperresolution generalizes ordinary hyper-
resolution, for one of the negative literals of clause (J-10), instead of being
removed with a positive clause, is removed with a clause containing a negative
literal, which in turn is removed with a positive clause. We note the parallel to the
preceding example (used to illustrate linked UR), which showed how the require-
ment to remove literals with the use of unit clauses was relaxed. For a second
parallel, just as the example of linked UR-resolution treats certain information as
present implicitly only, the example of linked hyperresolution treats the fact that
Roberta is not male as being present implicitly only.

Linked inference rules, as can be seen from the two examples, offer various
advantages over the corresponding standard versions and over logic programming.
First, information that would ordinarily be retained explicitly is instead retained
implicitly, an important property well recognized in logic programming. Second,
from the viewpoint of strategy (Section 2.3), the use of linked inference rules
permits one to make better choices for the clause or clauses on which to base the
entire attack on a given problem. In particular—in the spirit of the set of support
strategy (Section 2.3)—clause (J-1) is a far more effective choice than clause (J-5),
and clause (J-12) is a far more effective choice than clause (J-10).

In contrast, the use of standard UR resolution in the first example would
encourage one to key the attack on clause (J-5), and the use of logic programming
would force one to key the attack on clause (J-10). Clauses (J-5) and (J-10) are not
particularly satisfying choices because of their generality. As we discuss in the
section on strategy, the use of information that is specific to the problem is a far
better choice. Finally, rather than—as in logic programming—complete avoidance
of new information, the use of linked inference rules allows the user to specify the
types of intermediate information to retain. For example, the user can specify a
predicate, such as HASJOB, that is of interest, and have the program retain all
clauses in which the only predicate that occurs is that which is chosen by the user.

At this point, if one revisits the example of linked hyperresolution, one can see
how it does indeed generalize the type of reasoning employed in logic program-
ming. In both, one is allowed to use clauses containing both positive and negative
literals to remove a negative literal from the initiating nucleus or goal clause.
These mixed clauses link the goal clause to satellites—as they are called for
hyperresolution—or positive clauses. In other words, just as linked hyperresolution
gives the program the added latitude of using mixed clauses as links (which
parallels the added latitude in linked UR-resolution of allowing the use of nonunit
clauses as links), logic programming uses such mixed clauses.

Of course, since the retention of information is crucial to many applications of
automated theorem proving (in contrast to the single goal clause used in logic
programming), the use of linked hyperresolution will typically involve a number of
initiating nuclei or a number of nuclei each called into play because of the use of a
number of initiating satellites. Also, as already mentioned, logic programming does
not adhere to the spirit of hyperresolution (focusing on clauses as nuclei that
contain positive literals), while linked hyperresolution typically does, even allowing
the use of nuclei that are not Horn clauses. Nevertheless, one more connection
between logic programming and automated theorem proving exists, and—if one of
the many researchers in logic programming studies linked inference rules and
gains some as yet unseen insight—perhaps an extension to logic programming will
result that exhibits an even closer connection between the two fields.

2.3. Strategy

Despite our obvious enthusiasm about various inference rules and their potential
value to logic programming (for example, the use of paramodulation), we cannot
overemphasize the need for strategy to control their application. Indeed, by
applying the obvious metarule about getting nothing free, one can easily predict
that, when a powerful procedure (such as paramodulation) is used, hazards will be
encountered. The key is to use strategy—strategy to restrict reasoning, and
strategy to direct reasoning.

The need for strategy is clearly recognized in logic programming, as evidenced
by the use of a depth-first search strategy and a left-to-right ordering strategy for
solving subgoals. However, we suspect that very few people in logic programming
know that the field relies on a very restricted use of a well-known strategy in
automated theorem proving, the set of support strategy, that was introduced in 1965
[40]. In this section, we focus heavily on that strategy, and discuss it in its most
general form. We shall also discuss a strategy—the weighting strategy [17]—that
permits the user to impose knowledge and intuition on an automated theorem-pro-
vring program’s attack on a problem. Perhaps researchers in logic programming can
find a way to use the set of support strategy more fully than it is used now, and can
also find a way to use the weighting strategy. Certainly—from our viewpoint—the
current use of strategy in logic programming appears to be insufficient, if the goal
is to use logic programming to answer deep questions or solve hard problems from
mathematics. Indeed, perhaps the most significant issue raised in this article is that
focusing on the use and importance of strategy.
In fact, even with the power offered by an inference rule such as paramodulation—which enables a theorem-proving program to treat equality as “understood”—the use of strategy is essential. Simply stated—from the viewpoint of automated theorem proving—without strategy, the program can deduce far too many conclusions and too easily proceed in an unprofitable direction. From the viewpoint of logic programming (especially if the question under study is deep), without a sophisticated use of strategy, a program written in, say, PROLOG can too easily pursue a huge number of paths before finding an answer, if indeed it ever does.

For a taste of the power of paramodulation that shows why the use of strategy of various kinds is absolutely necessary—even though an examination of our example may, for some, (mistakenly) weaken our case for treating equality as a “built-in” concept—let us briefly focus on the equation that gives the actions of the combinator $W$ in combinatory logic:

$$(Wx)y = (xy)y.$$  

If one applies reasoning in the spirit of equality substitution—more accurately, if one applies paramodulation—to $W$ with itself, one deduces the following sixteen clauses:

$$(W(Wx)y)z = (((xy)y)z)z,$$
$$(Wx)(Wy)z = (x((yz)z)((yz)z),$$
$$(WW)x = (xx)x,$$
$$(W((xy)y))z = (((Wx)y)z)z,$$
$$(Wx)((yz)z) = (x((Wx)y)z)((yz)z),$$
$$(W(xy))y = ((Wx)y)y,$$
$$(Wx)((yz)z) = (x((yz)z)((Wx)z),$$
$$(Wx)y = (Wx)y, $$
$$(W((xy)y))z = (((Wx)y)z)z,$$
$$(Wx)((yz)z) = (x((Wx)y)z)((Wx)z),$$
$$(xy)y = (xy)y,$$
$$(W(Wx)y)z = (((xy)y)z)z,$$
$$(Wx)((Wy)z) = (x((yz)z)((Wy)z),$$
$$(W(Wx))y = ((xy)y)y, $$
$$(Wx)((Wy)z) = (x((Wx)y)z)((yz)z),$$
$$(WW)x = (xx)x.$$  

Of the sixteen given clauses, the first eight are obtained by paramodulating from the right side of the equation for $W$ into various terms, and the last eight are obtained by paramodulating from the left side. One can imagine how many additional conclusions would be drawn were we to continue reasoning by focusing on pairs of those sixteen conclusions—the members of the first generation of descendants of $W$ with itself—and then reason by focusing on the second generation.

To avoid the flood of conclusions that can result if reasoning is uncontrolled, an automated theorem-proving program can use various types of strategy. Some types of strategy are designed to restrict the reasoning; some are designed to direct it. From a different perspective, some types of strategy are general in the sense that they apply to all inference rules, and others are specific in the sense that they apply only to a single inference rule.
Of those of a general nature, the most powerful restriction strategy that an automated theorem-proving program can use is called the set of support strategy [40]. That strategy restricts the program's reasoning with the intention of sharply retarding the generation of irrelevant conclusions. To use the set of support strategy, one is required to choose a subset \( T \) of the set \( S \) of input clauses given to the program to describe the problem to be studied. The program then draws a conclusion only if it is recursively traceable to \( T \). Therefore, at the beginning of the program's attack on a problem—from a procedural viewpoint—an inference rule is applied to a subset of clauses from \( S \) only if at least one of its members is in \( T \). Any clause that is retained is automatically given the power of the clauses that are members of \( T \). Thus, after additional clauses have been retained, an inference rule can be applied to a subset of clauses if one of them is a clause not in \( S \)—a clause that is not an input clause. Stated another way, with the set of support strategy, the program is not allowed to apply an inference rule to a subset of clauses all of which are in \( S - T \).

To see that logic programming does indeed rely on the use of a restricted form of the set of support strategy, let us consider the recommended ways to use this strategy. We recommend either of two uses. For each, let any given set of clauses that is used to present a question or problem to a theorem-proving program be divided into three sets: (1) those that correspond to the axioms of the underlying theory and any lemmas that are to be used, (2) those (the special hypotheses) that correspond to the extra assumptions (if any), and (3) those (the denial of the conclusion) that correspond to assuming that the theorem is false. For example, let us consider the theorem that asserts that, in a group, if the square of every element is the identity, then the group is commutative. By applying our rule for partitioning the set of clauses used to present a problem, set (1) would consist of those clauses that correspond to the axioms of a group and any lemmas to be used, set (2) would consist of those clauses that correspond to the assertion that the square of every element is the identity, and set (3) would consist of those clauses that correspond to assuming that the group is not commutative.

Our first recommendation for the use of the set of support is to choose for the set \( T \) (see the informal definition given earlier) the union of the special hypothesis and the denial of the conclusion. Our second—and less preferred—recommendation is to choose for the set of support \( T \) only those clauses that correspond to the denial of the conclusion of the theorem under study. One should note that, since the clauses that are present because of assuming the theorem false may in fact include clauses free of \( \neg \) (not), even by following our second recommendation one does not necessarily imitate precisely what occurs in logic programming. The reason is obvious—in logic programming, the goal clause must be negative. For example, the goal clause for the illustration of linked hyperresolution (Section 2.2.3) would be

\[(J-10) \quad \neg \text{FEMALE}(x) \cup \neg \text{MALE}(x)\]

if logic programming were given this puzzle fragment to solve. As commented, such a choice is unfortunate in that it permits a program to delve deeply into background information rather than focusing on the specific program to solve—in particular, such a choice does not reflect the spirit of the set of support strategy.
We can now summarize to see how logic programming does indeed rely on the use of the set of support strategy. One simply adds a single clause—the goal clause—all of whose literals are in effect negative literals, and requires that all of the program's reasoning be recursively traceable to that single clause. Phrased in terms of automated theorem proving, one chooses for the set of support a single clause containing no positive literals. Ordinarily, the clause selected for the set of support would be one of the clauses that are present because of assuming the theorem under consideration false. However, for those cases in which each of those clauses contains at least one positive literal, the actions one is forced to take in logic programming are not in the spirit of automated theorem proving. In particular, in logic programming one must try various clauses from the problem description, requiring, of course, that each be in effect a negative clause. By taking such an action, one is still relying on the use of the set of support strategy, for the program is not allowed to apply its reasoning to a set of clauses none of which is traceable to the selected clause. Unfortunately, from the viewpoint of automated theorem proving and the intent of the set of support strategy, this action risks the danger of using an axiom as the set of support. We shall return to this issue later in this article.

At present, if one is contemplating the study of deep questions or hard problems by using an approach that closely imitates that taken in logic programming, the following analysis might merit serious thought. In the best case—if viewed from the perspective of the set of support strategy—the reasoning employed by logic programming mirrors that applied by an automated theorem-proving program when the assumption that the theorem is false leads to the addition of exactly one clause, and that clause consists of all negative literals. In the worst case—using a set of support that consists of a single clause and one that corresponds to an axiom of the theory from which the problem is taken—the reasoning employed by logic programming diverges sharply from that employed by a theorem-proving program. For deep questions or hard problems, such a choice for the set of support is at best unwise. Indeed, when the difficulty is high, the probability is also high that the program, to succeed, will be forced to pursue a number of paths more or less independently.

The need to pursue independent paths is tightly coupled with access to the presence in the initial set of support of at least a few clauses—certainly more than one. More important, such serious studies must not be burdened with reasoning that stems from the axioms of a theory. Rather, the needed reasoning must be severely restricted to the information outside of that which corresponds to axioms and lemmas. This last observation explains why our recommended use of the set of support strategy asks the user to place in the set of support, in addition to the clauses that are present because of assuming the theorem false, those that correspond to the special hypothesis.

If one were to follow our recommendation, and if one were to attempt to use a program to prove the commutator theorem relying on the P-formulation (Section 2.6), then one would place in the set of support clauses (COM-1) through (COM-8). In contrast, if one were to attack this theorem in the spirit of logic programming, then the set of support would consist of clause (COM-8) only. With this choice—rather than even following our second recommendation concerning
the set of support strategy—one would be using only part of the denial of the conclusion. Indeed, the denial consists of clauses (COM-3) through (COM-8). Although any attempt to prove the commutator theorem should prove exciting (playing against long odds), one might attempt to obtain a computer proof by forcing the set of support to consist of clause (COM-8) only. We conjecture that disaster will occur.

If, instead of relying on the $P$-formulation, one were to attempt to prove the commutator theorem using the notation that emphasizes equality, then one would use clauses (COMEQ-1) through (COMEQ-3), if the intent was to follow our preference. On the other hand, were one using a theorem-proving program based on logic programming technology, then one would use clause (COMEQ-3) only. One might find it challenging to make such an attempt. Again we conjecture that disaster will occur. Our conjecture becomes even more piquant when one notes that we did in fact succeed in proving the commutator theorem with one of our programs—the program $\text{TPO}$, which executes rather slowly compared to our other programs—and with only clause (COMEQ-3) in the set of support; however, the program relied on the use of the inference rule negative hyperparamodulation [36]. The proof was obtained in 193 CPU seconds on a Sun 3/75, generating 405 clauses and retaining 115. In contrast—and we recognize that comparative analyses are complicated and that the ideal data are unavailable—by using paramodulation and following our recommendation for the use of the set of support strategy, a proof was obtained with an earlier program in 3 CPU seconds on a computer that is five to ten times more powerful.

Although we heartily endorse a variety of approaches to serious studies and often learn from different paradigms for automated theorem proving, we regretfully conjecture that—without substantial data to the contrary—an approach to studies focusing on deep questions from mathematics, if based primarily on logic programming technology, must fail. Of course, for simple theorems such a technology may indeed prove fruitful.

Because one should attempt to at least provide an example when using a nebulous concept such as simple theorem, let us immediately do so. We include among simple theorems that which asks one to prove commutativity for groups for which one knows that the square of every element is the identity. This theorem was proved in 1965 with binary resolution and the set of support strategy and without demodulation (see Section 2.4) on a Control Data Corporation 3600 in less than 3 CPU seconds. The theorem is referred to as a classroom exercise by algebraists, but it does serve well for illustrating concepts and approaches. In contrast, we include among somewhat difficult theorems to prove the commutator theorem, which in Section 2.6 we present, discuss, and prove in two different ways.

We consider it important to exercise great care when estimating the power of an automated theorem-proving program, especially when the estimate is based on proving simple theorems. On the other hand, if one has designed a program that proves simple theorems in very little CPU time (say, less than 5 CPU seconds per proof) and with a high degree of success, then one is indeed justified in claiming an important achievement.

To complement the objective of the set of support strategy—for a general strategy that can be used to direct, rather than restrict, the program's reasoning—one can use the weighting strategy. To use this strategy, one assigns
priorities, called *weights*, to the various concepts, relations, and terms. The weights can be based simply on symbol count, or they can be based on a set of elaborate templates. The theorem-proving program uses the weights to choose from among the various clauses that on which to focus its attention. In other words, by using the weights to select which path of reasoning to follow, the program directs its reasoning.

For example, for the commutator theorem, one could instruct the program to prefer expressions containing one occurrence of the function inv to all other expressions; one simply assigns a low weight to inv and higher weights to all other functions and constants. Similarly, in the "jobs puzzle" and its variants, one can assign a low weight to Roberta, a slightly higher weight to Thelma, and the like, if one wishes the program to focus on information about Roberta in preference to focusing on information about Thelma. For a more detailed discussion of using weighting and the set of support strategy, one can browse in [38] or in [35].

In addition to using the set of support strategy, which is a general strategy designed to restrict the application of any inference rule, an automated theorem-proving program can also use a number of other strategies that are specific in that they are designed to restrict the application of paramodulation. First, the program can be restricted from paramodulating from or into a variable. This restriction prevents a myriad of conclusions from being drawn. It is, in most cases, mandatory. After all, paramodulating from or into a variable will always yield a conclusion, since the corresponding unification can never fail. Second, the program can be required to paramodulate only from a specified side of each equality. In some cases, all applications of paramodulation are required to be from the left side only; in others, from the right only.

Because of our desire to further pique the interest of researchers in logic programming, and because we explicitly recognize that far more control over reasoning is required for automated theorem proving to reach its potential, we close this section on strategy with a few incomplete notions about additional possibilities. One strategy that might merit exploration concerns choosing the path for an attempt at solving a subgoal according to the number of literals in the clause to be used, with a preference for unit clauses. A second strategy focuses on choosing the subgoal to attempt to solve according to the likelihood of failure. Either of the two strategies could take into account or ignore the current partial instantiation. A third strategy—which might be considered as a generalization of "assert" in PROLOG—would permit a user to give criteria for the program to apply to decide which items of information to retain. For a fourth strategy—one that might prove useful for enabling logic programming to rely more on a natural representation, as occurs to some degree in the theorem on index 2 (Section 2.1)—one could consider the use of automated tautology adjunction followed by case splitting. For example, if we borrow from an approach commonly taken in mathematics, the two natural cases to consider for the index-2 theorem focus, respectively, on the case in which $a$ is an element of the subgroup $O$ and on the case in which $a$ is not an element of the subgroup. As one no doubt guesses, currently it is not clear to us how to complete this approach to cope with non-Horn clauses. Nevertheless, because the use of strategy is vital to attacking deep questions and hard problems, consideration of the preceding incomplete notions as well as the others presented in this section might prove challenging.
2.4. Demodulation

The next topic for this review of automated theorem proving is that of canonicalization, which is available to automated theorem-proving programs through the use of demodulation [41] (see the two proofs in Section 2.6 below). In many areas of mathematics and logic, one often rewrites each new item of information into a canonical form. In automated theorem proving, a similar action can be taken. Of course, if as in logic programming no new clauses are retained, then one might conclude that canonicalization is of no interest. We suspect that such a position has hidden flaws in it, as we discussed in Section 2.2.3. Indeed, not only is demodulation essential for automated theorem proving; its use might also merit serious consideration for logic programming, which we briefly discussed in Section 2.2.3.

With regard to canonicalization, a sharp difference, however, exists between mathematics and automated theorem proving. In the former, one uses canonicalization when it seems convenient and appropriate. In the latter, one cannot in most cases exercise such freedom, for without automatically requiring each conclusion to be rewritten into a canonical form, the program will drown in information. Of course, some conclusions will be unaffected by the particular choice of demodulators [41] in a given problem.

One other difference exists between the use of canonicalization by a mathematician or logician and its use by a theorem-proving program. When a person uses various equalities for canonicalization, the equalities are applied only to the expressions chosen by the person; when a program applies such equalities, all expressions to which the equalities are applicable are in fact rewritten into a canonical form. Now, let us examine some typical uses of canonicalization.

One might, for example, uniformly replace \(-(-t)\) by \(t\) for any term \(t\), or replace \(r(s + t)\) by \(rs + rt\) for terms \(r\), \(s\), and \(t\). For such term replacements, many automated theorem-proving programs offer the mechanism demodulation, referred to earlier, which automatically rewrites expressions into a canonical form according to some set of rewrite rules called demodulators. The demodulators can be included among the input clauses, or the program can be asked to find potentially useful ones. For the two given examples, the demodulators

\[
\text{EQUAL}(\text{minus}(\text{minus}(x)),x)
\]

and

\[
\text{EQUAL}(\text{prod}(x,\text{sum}(y,z)),\text{sum}(\text{prod}(x,y),\text{prod}(x,z)))
\]

can be used, respectively.

As one might predict, care must be taken in the choice of which equality clauses to attempt to use as demodulators. In particular, an unwise choice can cause the program to loop. To see how looping can occur, let us consider what will happen when the equation for the combinator \(W\) is used as a demodulator and when the program encounters a term of a certain type. In particular, when the clause

\[
(W) \text{EQUAL}(a(a(W,x),y),a(a(x,y),y))
\]

is used as a demodulator and the term \(WWW\) is encountered, this term will be transformed to itself repeatedly without ever terminating. For a more interesting
example, let us consider the expression
\[
\Gamma = a(a(W,a(B,x)),a(W,a(B,x)))
\]
and what occurs when the two demodulators (for the combinators \(W\) and \(B\))
\[
(W) \text{ EQUAL}(a(a(W,x),y),a(x,y),y)
\]
and
\[
(B) \text{ EQUAL}(a(a(B,x),y),z),a(x,a(y,z))
\]
are applied to \(\Gamma\) with the object of producing a canonical form. Demodulation
begins by successfully applying \(W\), then \(B\), to obtain
\[
a(x,\Gamma),
\]
which leads to another successful application of \(W\) followed by one of \(B\) to obtain
\[
a(x,a(x,\Gamma)),
\]
which of course leads to yet another success and, in fact, to a nonterminating
sequence of successful applications of \(W\) and \(B\).

Nevertheless, as long as one is careful about the choice of demodulators, the
use of demodulation can sharply increase the effectiveness of a theorem-proving
program in its search for assignment completion. Further, for many studies,
without the use of demodulation the program will simple require an inordinate
amount of computer time to complete rather simple assignments.

For those familiar with functional programming, demodulation is similar to
reduction in which the reduction rules are the demodulators. In fact, demodulation
(in the context of automated theorem proving) has been used as a general-purpose
functional programming language [32].

2.5. Subsumption

We now come to the next-to-the-last topic of this section, subsumption, a mecha-
nism that—like demodulation—is almost always essential for a theorem-proving
program to be effective. Such a mechanism might also merit study for logic
programming, as we briefly touched on in Section 2.2.3. Although the use of
various types of strategy does indeed contribute markedly to the efficiency of an
automated theorem-proving program, such a program continues to deduce conclu-
sions that are trivial and unneeded corollaries of existing clauses. For example, if
the program paramodulates from the second argument of the clause equivalent of
\[
(W_x)y = (xy)y
\]
into the first subargument of the second argument of the clause equivalent of
\[
(W(W_x))y = ((xy)y)y,
\]
the program deduces the clause equivalent of
\[
(W(W_x))y = ((W_x)y)y,
\]
which is a trivial corollary, in fact an instance, of
\[
(W_x)y = (xy)y
\]
and is therefore immediately discarded. The procedure for discarding such deductions is subsumption [26]. The clause A subsumes the clause B if there exists a uniform replacement of terms for variables in A that produces a clause that is a subclause of B.

For a second example of subsumption, an example that might not be expected, let us consider the two clauses

\[ P(x) \mid P(y) \]

and

\[ \neg P(x) \]

and recall that a variable is relevant only to the clause in which it occurs. That the second clause subsumes the first follows immediately from the definition of subsumption. However, as it turns out, the first also subsumes the second, as can be seen by replacing y by x in the first to produce

\[ P(x), \]

since a clause, by definition, does not allow duplicate literals; technically, a clause is defined as a set of distinct literals. As an aside, if we wished to have our program use instantiation as an inference rule to permit the program to imitate the corresponding type of reasoning (discussed in Section 2.2) that is often used in mathematics, we would be forced to protect the instances that were deduced so that subsumption would not immediately discard them.

Of especial use is the role of subsumption for removing clauses that assert the equality of a term with itself; such an assertion is, needless to say, usually of little interest and seldom useful. To remove such trivial clauses, the clause

\[ \text{EQUAL}(x,x) \]

for reflexivity of equality is used to subsume many clauses that would otherwise be kept. This clause, which must be included in the input if paramodulation is the chosen rule of inference, is also important in that a proof often terminates with the deduction of a clause of the form

\[ \neg \text{EQUAL}(t,t) \]

for some term t.

2.6. Examples of Proof by Contradiction

In the interest of addressing all of the major elements of automated theorem proving, here we focus on proof by contradiction. We content ourselves with two proofs of a theorem, the commutator theorem, from group theory. This focus fulfills our promise, made in Section 1.3, of supplying proofs for the challenge problem posed there. For other interesting examples of such proofs, we recommend the two books cited earlier [38,35]. A study of proofs of theorems that are not considered trivial can be most instructive and, more important, can lead to an appreciation of what has been achieved, why certain mechanisms are used, and what is needed but not yet available.

Consideration of the commutator theorem illustrates (1) the need for using demodulation, (2) the need for using subsumption, (3) the need for retaining
clauses, and (4) the need for using powerful strategies. In short, the theorem we now consider—whose proof we give so that one can measure the performance of one’s program—provides an example of why we conjecture that shortcuts to producing a program that offers sufficient power must fail.

Although we ordinarily prefer a notation that emphasizes the role of equality, for this theorem we shall rely instead on the so-called $P$-formulation. One might note that the clauses we give for this problem are in fact Horn clauses, which means that the theorem is—theoretically—in reach of logic programming. One need not be familiar with group theory to study this theorem, especially since we give its proof. However, one’s understanding is increased by noting that the commutator $[x,y]$ of any two elements $x$ and $y$ of a group is the product of $x$, $y$, the inverse of $x$, and the inverse of $y$. The theorem to prove asserts that if the cube of every element $x$ in the group is the identity $e$, then $[[x,y],y] = e$ for all $x$ and $y$.

The following set of twenty-five clauses includes a subset that provides a complete characterization for abstract group theory—the first seventeen. Those twenty-five clauses can be used to present one notational version of the theorem to be proved. For those interested in the fine points of a theory, one can dispense with ten of the first seventeen clauses if one wishes to rely on a set containing no dependent clauses; indeed, the set consisting of clauses (GT-1), (GT-3), (GT-5), (GT-6), (GT-7), (GT-8), and (GT-14) also provides a complete characterization for abstract group theory.

Left identity:
(GT-1) $P(e,x,x)$

Right identity:
(GT-2) $P(x,e,x)$

Left inverse:
(GT-3) $P(inv(x),x,e)$

Right inverse:
(GT-4) $P(x,inv(x),e)$

Associativity:
(GT-5) $\neg P(x,y,u) \land \neg P(y,z,v) \land P(u,z,w) \land P(x,u,w)$

(GT-6) $\neg P(x,y,u) \land \neg P(y,z,v) \land \neg P(x,u,w) \land P(u,z,w)$

Closure (perhaps, more accurately, totality):
(GT-7) $P(x,y,\text{prod}(x,y))$

Product is well defined:
(GT-8) $\neg P(x,y,u) \land \neg P(x,y,v) \land \text{EQUAL}(u,v)$

Equality is reflexive:
(GT-9) $\text{EQUAL}(x,x)$

Equality is symmetric:
(GT-10) $\neg \text{EQUAL}(x,y) \land \text{EQUAL}(y,x)$

Equality is transitive:
(GT-11) $\neg \text{EQUAL}(x,y) \land \neg \text{EQUAL}(y,z) \land \text{EQUAL}(x,z)$
Equality substitution:

\( \text{(GT-12)} \quad \neg \text{EQUAL}(u,v) \rightarrow \neg P(u,x,y) \rightarrow P(v,x,y) \)

\( \text{(GT-13)} \quad \neg \text{EQUAL}(u,v) \rightarrow \neg P(x,u,y) \rightarrow P(x,v,y) \)

\( \text{(GT-14)} \quad \neg \text{EQUAL}(u,v) \rightarrow \neg P(x,y,u) \rightarrow P(x,y,v) \)

\( \text{(GT-15)} \quad \neg \text{EQUAL}(u,v) \rightarrow \text{EQUAL}(\text{prod}(u,x),\text{prod}(v,x)) \)

\( \text{(GT-16)} \quad \neg \text{EQUAL}(u,v) \rightarrow \text{EQUAL}(\text{prod}(x,u),\text{prod}(x,v)) \)

\( \text{(GT-17)} \quad \neg \text{EQUAL}(u,v) \rightarrow \text{EQUAL}(\text{inv}(u),\text{inv}(v)) \)

The cube of every \( x \) is the identity \( e \):

\( \text{(COM-1)} \quad \neg P(x,x,y) \rightarrow P(y,x,e) \)

\( \text{(COM-2)} \quad \neg P(x,x,y) \rightarrow P(x,y,e) \)

Denial of the conclusion:

\( \text{(COM-3)} \quad P(a,b,c) \)

\( \text{(COM-4)} \quad P(c,\text{inv}(a),d) \)

\( \text{(COM-5)} \quad P(d,\text{inv}(b),h) \)

\( \text{(COM-6)} \quad P(h,b,j) \)

\( \text{(COM-7)} \quad P(j,\text{inv}(h),k) \)

\( \text{(COM-8)} \quad \neg P(k,\text{inv}(b),e) \)

For the first proof we shall give, we use hyperresolution \cite{27}, a set of demodulators in the input, and demodulators that are found by the program during its search for a proof. The demodulators found by the program are used to back-demodulate (rewrite) clauses already present, as well as demodulate clauses deduced after the program adjoins these new demodulators. In the following proof, the clause numbers are taken directly from the successful proof search.

**PROOF 1 OF THE COMMUTATOR THEOREM**

Closure (totality):

\( \text{(PCOM-1)} \quad P(x,y,\text{prod}(x,y)) \)

Left inverse:

\( \text{(PCOM-4)} \quad P(\text{inv}(x),x,e) \)

Right inverse:

\( \text{(PCOM-5)} \quad P(x,\text{inv}(x),e) \)

Associativity:

\( \text{(PCOM-6)} \quad \neg P(x,y,u) \rightarrow \neg P(y,z,u) \rightarrow \neg P(u,z,w) \rightarrow P(x,v,w) \)

\( \text{(PCOM-7)} \quad \neg P(x,y,u) \rightarrow \neg P(y,z,u) \rightarrow \neg P(x,v,w) \rightarrow P(u,z,w) \)

Product is well defined:

\( \text{(PCOM-11)} \quad \neg P(x,y,u) \rightarrow \neg P(x,y,v) \rightarrow \text{EQUAL}(u,v) \)

Left cancellation:

\( \text{(PCOM-18)} \quad \neg P(x,u,y) \rightarrow \neg P(x,v,y) \rightarrow \text{EQUAL}(u,v) \)

(Another form of) the cube of \( x \) is \( e \):

\( \text{(PCOM-20)} \quad P(x,\text{prod}(x,x),e) \)

Denial of the conclusion:

\( \text{(PCOM-21)} \quad P(a,b,c) \)

\( \text{(PCOM-22)} \quad P(c,\text{inv}(a),d) \)

\( \text{(PCOM-24)} \quad P(h,b,j) \)

\( \text{(PCOM-25)} \quad \neg P(k,\text{inv}(b),e) \)
Demodulators:

(PCOM-27) \( \text{EQUAL}(\text{prod}(e,x),x) \)

(PCOM-28) \( \text{EQUAL}(\text{prod}(x,e),x) \)

(PCOM-31) \( \text{EQUAL}(\text{prod}(\text{prod}(x,y),z),\text{prod}(x,\text{prod}(y,z))) \)

(PCOM-37) \( \text{EQUAL}(\text{prod}(a,b),c) \)

(PCOM-40) \( \text{EQUAL}(\text{prod}(h,b),d) \)

(PCOM-41) \( \text{EQUAL}(\text{prod}(d,\text{inv}(h)),k) \)

From clauses 21, 6, 1, and 1,

(PCOM-42) \( P(a,\text{prod}(b,x),\text{prod}(c,x)) \)

From clauses 21, 6, 4, and 1, with demodulation by 27,

(PCOM-48) \( P(\text{inv}(a),c,b) \)

From clauses 24, 11, and 1, with demodulation by 40,

(PCOM-58) \( \text{EQUAL}(d,j) \)

From clause 58 back-demodulating clause 41,

(PCOM-68) \( \text{EQUAL}(\text{prod}(j,\text{inv}(h)),k) \)

From clause 58 back-demodulating clause 22,

(PCOM-70) \( P(c,\text{inv}(a),j) \)

From clauses 48, 11, and 1,

(PCOM-104) \( \text{EQUAL}(\text{prod}(\text{inv}(a),c),b) \)

From clauses 70, 7, 4, and 1, with demodulation by 28,

(PCOM-137) \( P(j,a,c) \)

From clauses 70, 6, 21, and 1,

(PCOM-147) \( P(a,\text{prod}(b,\text{inv}(a)),i) \)

From clauses 20, 6, 1, and 1, with demodulation by 27 and 31,

(PCOM-190) \( P(x,\text{prod}(x,\text{prod}(x,y)),y) \)

From clauses 20, 18, and 5,

(PCOM-199) \( \text{EQUAL}(\text{inv}(x),\text{prod}(x,x)) \)

From clauses 20, 6, 24, and 1,

(PCOM-211) \( P(h,\text{prod}(b,\text{prod}(j,j)),e) \)

From clauses 20, 6, 137, and 1,

(PCOM-213) \( P(j,\text{prod}(a,\text{prod}(c,c)),e) \)

From clause 199 back-demodulating clause 147,

(PCOM-224) \( P(a,\text{prod}(b,\text{prod}(a,a)),j) \)

From clause 199 back-demodulating clause 104, with demodulation by 31,

(PCOM-241) \( \text{EQUAL}(\text{prod}(a,\text{prod}(a,c)),b) \)

From clause 199 back-demodulating clause 48,

(PCOM-242) \( P(\text{prod}(a,a),c,b) \)

From clause 199 back-demodulating clause 26,

(PCOM-265) \( \neg P(k,\text{prod}(b,b),e) \)

From clause 199 back-demodulating clause 68,

(PCOM-285) \( \text{EQUAL}(\text{prod}(j,\text{prod}(h,h)),k) \)
From clauses 242, 6, 1, and 1,
(\text{PCOM-375}) \ EQuAL(\text{prod}(a, \text{prod}(a, c), b))

From clauses 42, 11, and 1,
(\text{PCOM-498}) \ EQuAL(\text{prod}(a, \text{prod}(b, x)), \text{prod}(c, x))

From clauses 211, 18, and 20,
(\text{PCOM-1092}) \ EQuAL(\text{prod}(b, \text{prod}(j, j)), \text{prod}(h, h))

From clauses 213, 18, and 20,
(\text{PCOM-1117}) \ EQuAL(\text{prod}(a, \text{prod}(c, c)), \text{prod}(j, j))

From clauses 224, 7, 190, and 375, with demodulation by 31, 31, 241, 37, 31, 31, 498, 1117, and 1092,
(\text{PCOM-1161}) \ P(j, \text{prod}(h, h), b)

From clauses 1161, 11, and 1, with demodulation by 285,
(\text{PCOM-1214}) \ EQuAL(b, k)

From clause 1214 back-demodulating clause 265 twice,
(\text{PCOM-1589}) \ \neg P(k, \text{prod}(k, k), e)

Clause (\text{PCOM-1589}) contradicts clause (\text{PCOM-20}), and the proof is complete.

For experiments that produce proofs like the preceding and like the following, we prefer to employ some or all of the following clauses as demodulators. This set of ten clauses is a complete set of reductions for group theory—more accurately, for free groups (see Chapter 5 of [35]). For the theorem under discussion, we used the first seven only:

\begin{align*}
EQuAL(\text{prod}(e, x), x) \\
EQuAL(\text{prod}(x, e), x) \\
EQuAL(\text{prod}(\text{inv}(x), x), e) \\
EQuAL(\text{prod}(x, \text{inv}(x)), e) \\
EQuAL(\text{prod}(\text{prod}(x, y), z), \text{prod}(x, \text{prod}(y, z))) \\
EQuAL(\text{inv}(\text{inv}(x)), x) \\
EQuAL(\text{inv}(e), e) \\
EQuAL(\text{inv}(\text{prod}(x, y)), \text{prod}(\text{inv}(y), \text{inv}(x))) \\
EQuAL(\text{prod}(\text{inv}(x), \text{prod}(x, y)), y) \\
EQuAL(\text{prod}(x, \text{prod}(\text{inv}(x), y)), y)
\end{align*}

Let us stay with the commutator theorem just a bit longer to show how it can be proved by using paramodulation and an equality-oriented notation. To study abstract group theory, but with an emphasis on the use of equality, one can use the following clauses:

Left identity:
(\text{GTEQ-1}) \ EQuAL(\text{prod}(e, x), x)

Right identity:
(\text{GTEQ-2}) \ EQuAL(\text{prod}(x, e), x)
Left inverse:
(GTEQ-3) \( \text{EQUAL}(\text{prod}(\text{inv}(x), x), e) \)

Right inverse:
(GTEQ-4) \( \text{EQUAL}(\text{prod}(x, \text{inv}(x)), e) \)

Associativity:
(GTEQ-5) \( \text{EQUAL}(\text{prod}(\text{prod}(x, y), z), \text{prod}(x, \text{prod}(y, z))) \)

Reflexivity:
(GTEQ-6) \( \text{EQUAL}(x, x) \)

One might note that—as expected—the only axiom for equality itself that we include is that for reflexivity.

To have an automated theorem-proving program attempt to prove the commutator theorem, one can add the following clauses:

The cube of every \( x \) is the identity \( e \):
(COMEQ-1) \( \text{EQUAL}(\text{prod}(x, \text{prod}(x, x)), e) \)

Definition of commutator:
(COMEQ-2) \( \text{EQUAL}(\text{com}(x, y), \text{prod}(x, \text{prod}(y, \text{prod}(\text{inv}(x), \text{inv}(y))))) \)

Denial of the conclusion:
(COMEQ-3) \( \neg \text{EQUAL}(\text{com}(\text{com}(a, b), b), e) \)

So that one can study how paramodulation works on a hard problem, so that one can see how effective this inference rule is, and so that one can measure one's experiments against what we have obtained, we give the following proof based on paramodulation. We use as input demodulators the same seven we used for the \( P \)-formulation.

**PROOF 2 OF THE COMMUTATOR THEOREM**

Associativity:
(PCOMEQ-5) \( \text{EQUAL}(\text{prod}(\text{prod}(x, y), z), \text{prod}(x, \text{prod}(y, z))) \)

Provable lemma in group theory:
(PCOMEQ-8) \( \text{EQUAL}(\text{inv}(\text{prod}(x, y)), \text{prod}(\text{inv}(y), \text{inv}(x))) \)

Provable lemma in group theory:
(PCOMEQ-10) \( \text{EQUAL}(\text{prod}(\text{inv}(x), \text{prod}(x, y))), y \)

Reflexivity:
(PCOMEQ-11) \( \text{EQUAL}(x, x) \)

The cube of \( x \) is the identity \( e \):
(PCOMEQ-12) \( \text{EQUAL}(\text{prod}(x, \text{prod}(x, x)), e) \)

Denial of the conclusion, after demodulation with definition of commutator:
(PCOMEQ-13) \( \neg \text{EQUAL}(\text{prod}(a, \text{prod}(b, \text{prod}(\text{inv}(a), \text{prod}(b, \text{prod}(a, \text{prod}(\text{inv}(b), \text{prod}(\text{inv}(a), \text{inv}(b))))))), e) \)

Demodulators:
(PCOMEQ-14) \( \text{EQUAL}(\text{prod}(e, x), x) \)
(PCOMEQ-15) \( \text{EQUAL}(\text{prod}(x, e), x) \)
(PCOMEQ-18) \( \text{EQUAL}(\text{prod}(\text{prod}(x, y), z), \text{prod}(x, \text{prod}(y, z))) \)

From clause 12 into clause 5, with demodulation by 18, 18, and 18,
(PCOMEQ-24) \( \text{EQUAL}(\text{prod}(x, \text{prod}(y, \text{prod}(x, \text{prod}(y, \text{prod}(x, y))))), e) \)
From clause 12 into clause 5, with demodulation by 18 and 14,
(PCOMEQ-28) \text{EQUAL}(\text{prod}(x, \text{prod}(x, \text{prod}(x, y))), y)

From clause 27 into clause 8, with demodulation by 27, 18, 27, and 18,
(PCOMEQ-30) \text{EQUAL}(\text{prod}(x, \text{prod}(x, \text{prod}(y, y))), \text{prod}(y, \text{prod}(x, \text{prod}(y, x))))

From clause 27 into clause 13, with demodulation by 27, 27, 18, 27, 18, 18, 30, and 28,
(PCOMEQ-31) \neg \text{EQUAL}(\text{prod}(a, \text{prod}(b, \text{prod}(a, \text{prod}(a, \text{prod}(b, \text{prod}(a, \text{prod}(b, a)))))), e)

From clause 24 into clause 5, with demodulation by 18, 18, and 18,
(PCOMEQ-33) \text{EQUAL}(\text{prod}(x, \text{prod}(y, \text{prod}(z, \text{prod}(x, \text{prod}(y, \text{prod}(z, \text{prod}(x, \text{prod}(y, z)))))))), e)

From clause 33 back-demodulating clause 31,
(PCOMEQ-35) \neg \text{EQUAL}(e, e)

Clause (PCOMEQ-35) contradicts clause (PCOMEQ-11), and the proof is complete. \qed

One who enjoys historical highlights might find it interesting to note that reading in 1970 about this problem and its binary resolution and paramodulation proofs (obtained by hand) motivated R. Overbeek to become interested in automated theorem proving. Overbeek obtained a computer proof with hyperresolution in 1975, a proof requiring 100 CPU seconds on an IBM 370/195 with a program he designed and implemented. With that same program on the same computer, its proof was obtained with paramodulation in 1976 in less than 3 CPU seconds.

A study of the two given proofs, or an attempt to obtain one of them, may amply demonstrate why automated theorem proving relies heavily on the use of the set of support strategy, demodulation, and subsumption. The study or the attempt may also provide one with excitement and a challenge. Regardless of the outcome, theorems like the commutator theorem provide, directly and indirectly, excellent illustrations of some of the fundamental differences between logic programming and automated theorem proving. Nevertheless, each field offers the other many important ideas to consider and techniques to adapt (discussed in Section 4), and a good case can be made for the existence of a natural symbiosis.

Having completed our lengthy review of automated theorem proving, having made certain comparisons with logic programming, and desiring to focus on that possible symbiosis, let us now turn to collecting some of the important similarities and some of the important differences between the two fields. This action will set the stage for discussing the possible benefits each field offers the other, which is the topic of Section 4.

3. SIMILARITIES AND DIFFERENCES

Although in some sense it may seem odd to compare logic programming and automated theorem proving—since they address fundamentally different issues, namely, programming language versus general-purpose deduction—nevertheless, we find such a comparison of interest. In particular, especially for those who are new to logic programming and automated theorem proving, the comparison may
remove certain points of confusion regarding the possible use of the former for the latter.

The most obvious similarity between logic programming and automated theorem proving—at least, if one focuses on the clause language paradigm—is the language that one uses to communicate a problem. Of course, the syntax may vary, and most dialects of logic programming require the use of Horn clauses. Nevertheless, both fields rely on the use of clauses. In automated theorem proving, however, some programs honor a "built-in" treatment of equality in the full sense of the term. For example, use of the predicate \texttt{EQUAL} causes such programs to "know" that the corresponding statement refers to equality. Because of our obvious vested interest, we are delighted that familiarity with logic programming is rapidly growing, so that eventually the vast majority of computer scientists will know what a clause is. Such knowledge will increase the likelihood of computer scientists using an automated theorem-proving program.

The second important similarity between logic programming and automated theorem proving rests with the fact that the type of reasoning employed by the former is one of the types that can be employed by the latter. That reasoning, at the top level, as we discussed in Section 2.2.3, corresponds to linked hyperresolution. At the lower level, every reasoning step of each path that solves a subgoal is a successful application of binary resolution.

In addition to sharing certain aspects of representation and inference rule, logic programming and automated theorem proving share certain aspects of the use of strategy. Specifically, both rely heavily on the use of the set of support strategy (see Section 2.3). However, logic programming uses a very restricted version of this strategy.

Finally, in the sense discussed in Section 2.6, both fields rely on proof by contradiction. Further, just as logic programming constructs or discovers the particular object or objects of concern, automated theorem proving can do the same by using the \texttt{ANSWER} literal. Of course—to be precise—logic programming in effect uses such a literal by its employment of the goal clause. In contrast, automated theorem proving, when desired, explicitly uses the \texttt{ANSWER} literal whose arguments, upon completion of the given assignment, contain the sought-after information.

With regard to the differences between logic programming and automated theorem proving, at least four crucial ones immediately come to mind, and a number of lesser ones can be given. First—in the typical approach—logic programming restricts representation by requiring that all clauses but the goal clause contain exactly one positive literal. This restriction is a tightening of that which asks for all clauses to be Horn clauses, clauses that contain no more than one positive literal. In automated theorem proving, no restriction is placed on the clauses to be used in presenting a question or a problem.

Second, in logic programming—and in theorem-proving based on a similar paradigm, for example [29]—no new clauses are retained during an attack on a question or problem. This lack of retention clearly is one important source of the impressive speed offered by this programming language. Automated theorem proving, on the other hand (with the exception of the program already mentioned and similar ones) relies heavily on the retention of new information. Although such retention (when all other considerations are ignored) can be expensive, we conjec-
ture that, especially for deep questions and hard problems, it is absolutely necessary for effectiveness: a point we address in Section 2.2.3. In fact, for those who doubt this necessity, we shall give some challenge problems in Section 5, in addition to our recommendation of attempting to prove the commutator theorem.

The third important difference focuses on the use of strategy. Although logic programming—as we have observed—does use strategy, its use is extremely limited in comparison with the rich and sophisticated use of strategy in automated theorem proving. Though the limitation of logic programming to a single search strategy contributes to performance (even more than avoiding the occurs check does), to confine the use of strategy to a depth-first and left-to-right search and to a very limited form of the set of support strategy appears to us to cheat logic programming of much of its potential. Indeed, for logic programming to dramatically widen its scope of application to include the consideration of deep questions from mathematics, we conjecture that a far more sophisticated use of strategy is required. For one example, logic programming might consider using a broader form of the set of support strategy (as discussed in Section 2.3). For a second example, giving logic programming access to a strategy such as the weighting strategy, which allows the user to give hints about what is important, also appears to us to be a move that would add to the effectiveness of logic programming. After all, since the intent in logic programming is to enable one to apply an algorithm and in the context of logical reasoning, it seems natural to also permit the user to give hints.

The fourth significant difference between logic programming and automated theorem proving is in their treatments of equality in its fullest meaning. From what we know, no satisfactory and fully general treatment currently exists in logic programming. If this is the case, an outside observer should understand why, for the obstacles presented by equality are clearly most formidable. Indeed, although we are very partial to the treatment given to equality by the use of the inference rule paramodulation, we openly recognize that this inference rule does not by itself solve all of the problems that accompany a natural treatment of equality. As a result, we are still actively searching for ever more powerful strategies to control the application of paramodulation.

In contrast to using paramodulation, logic programming—to study formal semantics—has considered building equality into unification [8], more or less in the style of E-resolution [20]. Such an approach, which goes even further than using associative-commutative unification, has actually been tried in automated theorem proving, and the practical results are very disappointing. A number of researchers in logic programming would in fact have predicted such an outcome, believing that building equality into unification has little practical value.

Given these observations, we conjecture that logic programming, to fulfill its potential, must address equality directly and in its full generality, either by using a form of paramodulation or by some other means.

Among the other noteworthy differences is the absence of the occurs check in most of logic programming, in contrast to its constant presence in automated theorem proving. We do understand and accept that its absence contributes in an important way to the excellent speed that is offered, but offering the option—not requirement—of using the occurs check seems to merit serious consideration. Other differences worth mentioning regard the use, in automated theorem proving,
of demodulation to produce canonical forms for information and the use of subsumption to remove information captured by more powerful statements. Of course, neither of these procedures is relevant when no new information is retained.

Finally, in the strictest sense—with the exceptions concerning deductive databases, expert systems, and limited aspects of theorem proving—logic programming is a programming language, designed to efficiently execute well-tuned algorithms, especially those in which logical reasoning plays an important role. In contrast, an automated theorem-proving program is a collection of procedures that, taken together, form a reasoning system designed to attack problems for which one does not know of an effective algorithm to apply. Unfortunately, the danger exists of confusing the two: logic programming and automated theorem proving.

In particular, one should not attempt to take a shortcut to producing a system with the needed power offered by some theorem-proving programs—a shortcut of the form in which a small modification is made to logic programming, or one inference rule is adjoined. After all, if one considers how impressively mathematicians and logicians apply logical reasoning to questions and problems, one should expect that no shortcut exists to producing a computer program that can compete with such powerful minds or that can be used to significantly assist such minds. For example— and we borrow from experience as poker players—simple strategies produce small results. Continuing with the analogy, if one’s play is governed solely by the probabilities that can be found in the appropriate tables, then one will win at poker only if the other players are very weak. The other players in the game of science—the open questions to answer and problems to solve—are not very weak; to win in a contest with them, heavy use of strategy—as well as other complex procedures—is required. In addition to our total conviction concerning the need to use strategy, we also strongly favor a computer-oriented approach to automated theorem proving—in contrast to a person-oriented approach that attempts, for example, to emulate some great mathematician.

However, despite our comments that some—unfortunately—might consider negative, we are convinced that each of the two fields, logic programming and automated theorem proving, offers the other a great deal. Therefore, let us now turn to that topic, and to the symbiosis that could exist—and, in fact, already does in part exist.

4. SYMBIOSIS: BENEFITS EACH FIELD OFFERS THE OTHER

As the titles of this article and section suggest, we conjecture that the two fields of automated theorem proving and logic programming, even more than complementing each other, offer each other various important features—to the point that a possible symbiosis exists. We shall be content with merely sampling what is possible, since we have made corresponding observations throughout this article.

With regard to a general way in which logic programming complements automated theorem proving, there exist many problems with the property that certain aspects of an attack on them should be treated algorithmically, while other aspects cannot. For example, in program verification and, even more so, in Gödel’s finite
The axiomatization of set theory (see Chapter 6 of [35]), certain literals in various input and various deduced clauses have the property that they are shared. Of these literals, many correspond to conditions that must be checked routinely to see if they are satisfied. One promising approach to increasing the effectiveness of an automated theorem-proving program would have logic programming address the removal of these literals in an algorithmic manner while the theorem-proving program itself applies the general, less focused reasoning to the remaining literals. As evidence that supports the feasibility of such an approach—although we in no way recommend the imitation of the way people solve problems—consideration of various proofs produced by a person (in contrast to those produced by a computer) suggests that some of what occurs is indeed algorithmic, but most of it cannot be treated algorithmically.

For an example of a more specific benefit logic programming offers to automated theorem proving (and here we draw on one of the implementation techniques found in OTTER, our newest theorem-proving program), demodulation can be far less expensive in CPU time if, instead of the commonly used approach of applying substitutions and copying clauses, one replaces those two operations with the manipulation of pointers, as occurs in logic programming. For an example based on conjecture, various resolution-based linked inference rules might be implemented by compiling the linking clauses and nuclei in the way that logic programs are compiled; the result might indeed offer one or two orders of magnitude faster execution. For examples of using logic programming technology as the basis for automated theorem proving in general, one might study the work of Stickel [29] or the work of Plaisted [24].

With regard to possible benefits that automated theorem proving offers logic programming—and, as expected, we can and do say more on this topic, for this is our field of expertise—two main areas merit investigation. The first, the treatment of equality, must be addressed in its fullest if logic programming is to fulfill the vision of its zealots (which in no way implies that this programming language is anything but impressive). Obviously, equality plays such a vital role in so many questions and problems that a means must be found to more than adequately cope with the obstacles equality presents. Perhaps a study of how equality is treated in the clause language paradigm—specifically, the ways discussed in Sections 2.2.2 and 2.4—will provide a clue, or even provide the key.

Our preference—as we have made abundantly clear—is to use the inference rule paramodulation. As evidence of its power, we cite our most recent successes in combinatory logic [19, 37]. Paramodulation, in addition to generalizing the usual notion of equality substitution, also permits a program to reason deep within nested expressions directly. Although logic programming often appears reluctant to use nested expressions (except, of course, where the expressions can be treated algorithmically and where equality is avoided), if one wishes to attack open questions and hard problems, one must expect to encounter deeply nested expressions. For equational programming and for a combination of equational and logic programming, certain aspects of paramodulation have already been proposed [8].

The second area of automated theorem proving whose study might offer researchers in logic programming substantial reward is that concerning the use of various types of strategy. As discussed in Section 2.3, the set of support strategy can be used to restrict the reasoning to conclusions that are recursively traceable.
to the information chosen to support the entire investigation, and the weighting strategy can be used to direct reasoning by permitting the user to play an active advice-giving role. In addition to other existing strategies that might be of use, perhaps researchers in logic programming can formulate new ones. If so, then researchers and implementors in automated theorem proving must examine the new results.

In other words, although our treatment is perhaps too brief, strong evidence exists for the position that the two fields complement each other. Even further, strong evidence exists that—even more than at the implementation level—the two fields offer each other ideas to consider, enough ideas to suggest that symbiosis between automated theorem proving and logic programming does exist.

5. CONCLUSIONS, RECOMMENDATIONS—AND CHALLENGES

Before turning to our conclusions and recommendations we close on a high note—suspecting that most researchers enjoy challenges—by presenting the following questions for logic programming and also for automated theorem proving. Let the set $P$ consist of two combinators $B$ and $N$ which, respectively, satisfy the following two equations:

$((Bx)y)z = x(yz)$.

$((Nx)y)z = ((xz)y)z$.

The first two challenge questions ask whether the set $P$ satisfies the weak or the strong fixed-point property.

**Definition.** Where $P$ is a given set of combinators, the weak fixed-point property holds for $P$ if and only if for all combinators $x$ there exists a combinator $y$ such that $y = xy$, where $y$ is expressed purely in terms of the combinators in $P$ and the variable $x$.

**Definition.** Where $P$ is a given set of combinators, the strong fixed-point property holds for $P$ if and only if there exists a combinator $y$ such that, for all combinators $X$, $yx = x(yx)$, where $y$ is expressed purely in terms of combinators in $P$.

In clause form, we can use

$\neg\text{EQUAL}(y,a(f,y))$

to deny that the weak fixed-point property holds, and we can use

$\neg\text{EQUAL}(a(y,g(y)),a(g(y),a(y,g(y))))$

to deny that the strong fixed-point property holds, where $f$ is a Skolem constant, and $g$ is a Skolem function.

Since we know the answers to these two questions (see [37]), and since one might prefer an open question to study, let us consider the set $P'$ consisting of the two combinators $B$ and $S$, where

$((Sx)y)z = (xz)(yz)$. 
The third and fourth challenge questions ask whether $P'$ satisfies the weak or the strong fixed-point property.

Since we have kept our promise concerning challenges by giving four questions to consider—note that many research questions focusing directly on automated theorem proving can be found in [35]—let us next turn to our view of the present and the future. We are more optimistic now than at any other time since 1963, when we first became interested in the automation of logical reasoning (the term automated reasoning was not introduced until seventeen years later). The advent of parallelism, the design of new theorem-proving programs such as OTTER, the occurrence of numerous advances in logic programming and in automated theorem proving, all tell the same story. The future is indeed bright, and the opportunities to contribute in a significant way are greater than at any time in the history of either field. Therefore, we recommend that, although the amount of experimentation is substantially greater than it was even five years ago, far more experiments be conducted.

We also recommend that researchers in logic programming strongly consider offering users some or all of the options we have discussed in this article, among which are the use of a variety of strategies, a built-in and practical treatment of equality, and access to using the occurs check by setting a flag. We even suggest that, since we wish to encourage the use of natural representation within the clause language paradigm, the restriction to Horn clauses be relaxed. Indeed, as we have been told by F. Pereira [22], such a relaxation would be of substantial value to those interested in language parsing and language translation. Obviously, we wish to encourage the corresponding difficult but potentially significant research focusing on the variety of topics discussed in this article, and—taking a big chance—we predict that many extensions of the type we have discussed will exist shortly.

Of the various options discussed in this article, we note that the occurs check must be offered—or some other guarantee of the soundness of the reasoning be given—if logic programming is to be used as a deductive system. Although all researchers in logic programming are fully aware of the potential for drawing an unsound conclusion, we were startled at how easy it was to construct the following entertaining example illustrating the need for the occurs check. First, one is given the assertion that for all $x$ there exists a $y$ with $x$ less than $y$. Then, unexpectedly, one is asked to prove that there exists an $x$ such that for all $y$, $y$ is less than $x$. For the first of the two statements, one can use the clause

$$\text{LESSTHAN}(x,f(x)),$$

and for the second, when assumed false, one can use the clause

$$\neg\text{LESSTHAN}(g(x),x),$$

where $f$ and $g$ are appropriate Skolem functions. Given these two clauses—if the occurs check is ignored—a program will decide that a proof by contradiction has been found. Unfortunately, such a discovery proves that the existence of ever larger integers—which almost everyone accepts as true—implies the existence of a largest integer (even larger than itself), which many people doubt.

To close this section (and we are concerned that the observations we are about to repeat may disappoint or, worse, may annoy too many people), certain positions
must be challenged, and challenged strongly. The most difficult for us to challenge—difficult because we have such great respect for some who hold the position—concerns a particular class of paradigm for automated theorem proving. Succinctly stated, we conjecture that one cannot design and implement an automated theorem-proving program based on the current paradigms for logic programming, if that program is to be used to attack deep questions or hard problems. We are not being self-contradictory, for we do feel that one can profitably borrow from logic programming. Our conjecture is based on twenty-five years of experimentation—we certainly would like to believe that we are not simply being old-fashioned—and on our conviction that a sophisticated use of strategy is required and the retention of significant clauses is mandatory. We in fact more fully addressed this issue in Section 2.3 and in Section 2.2.3, when we discussed the need for clause retention and for the use of both demodulation and subsumption.

The next position we must challenge is that which in effect asserts that small modifications or simple fixes will suffice to extend logic programming to a theorem-proving program that offers substantial power. The difficulty of interesting questions from mathematics and logic makes it virtually impossible for such an easy and quick solution to work. Also—and here we pay appropriate tribute to various system designers—Boyer and Moore, on the one hand, and Overbeek and company, on the other, have devoted and are devoting many hundreds of hours to the consideration of implementing a very powerful theorem-proving program. Researchers of their caliber could not have overlooked such a simple solution to the problem of implementation.

The third and final position we must challenge is that which implies that logic programming as it now stands offers sufficient power to attack deep questions and solve hard problems. Although we laud the achievements of this field, and although we expect to see far more impressive ones in the future, such a position is, in most cases, doomed to disappointment. After all, logic programming is primarily a programming language, and not a deep reasoning system.

However—hoping that the preceding comments have not caused too many to simply cease reading—we again note that the two fields do complement each other, and that each potentially offers the other a great deal. Among the offerings are certain implementation techniques, a rather pleasing treatment of equality, various types of strategy, and far more. For those researchers who prefer a definite position concerning which field offers which the most, we conclude from our experience that, currently, automated theorem proving offers more to logic programming than the reverse. More important than this comparison is the natural symbiosis that exists between the two fields and the pursuit of the common goal of an effective automation of logical reasoning.

REFERENCES


