A Primer for Automated Reasoning: Puzzles, People, and Programs That Involve Reasoning

Larry Wos

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
wos@mcs.anl.gov

1. High Adventure, A Frontier of Computing

Do you enjoy solving puzzles, playing chess, assisting others in careful reasoning, and—often much more difficult—finding proofs of theorems in mathematics or logic? Do you know that computer programs exist that can assist you in such activities in unexpected ways and, sometimes, find answers that have eluded fine minds for many decades? Of course you know that computers can cope with gigantic arithmetic problems and can store, process, and retrieve huge amounts of data. But how in the world can a computer be programmed to reason logically, to take a set of assumptions and use those assumptions to reach goals that might appear to be unreachable? Indeed, how can you program a computer to reason like Mr. Spock of Star Trek or like Sherlock Holmes? You may indeed have read of a computer program that played brilliant chess. But you may not know of a computer program that is so powerful in its reasoning that various areas of mathematics and logic have yielded to that program some of their well-kept secrets.

Would you enjoy winning the lottery, winning a poker hand, winning a bowling game, winning at a particular relationship? Did you just implicitly ask me if I play computer games that you have never played? Yes—well, just one computer game, a game called automated reasoning. You have guessed correctly: Here I tell you what automated reasoning is, how it works, why you might find its use exciting, and—so intriguing—how strategy comes into play. Do not fear; for the individual who wishes a sharp taste of automated reasoning, I offer you in Section 4 some real food for thought, big challenges indeed. You will also learn that, more than occasionally, you can have a reasoning program mirror the actions a person might take. However, as you will learn, the powerful automated reasoning programs can, and often do, reason in a manner that is sharply different from that of a person—relentlessly and tirelessly drawing (possibly) millions and millions of conclusions of which some millions are retained as the goal is reached.

For but one example, I shall introduce to you a method of reasoning that is awesomely effective, a rule that even great minds would not apply unaided. (For the curious, this inference rule is called paramodulation; a detailed treatment is given in Section 4.) You and a well-chosen automated reasoning program can team up in remarkable ways to use such rules. All will be presented in an informal manner, relying on various examples rather than a rigorous treatment. Especially if you like some area of mathematics or logic, but even if not, you will learn that serious research can be done, and is being done here in 2014, with the program I shall feature in this notebook, William McCune's OTTER. (A copy of OTTER can be obtained by consulting my website automatedreasoning.net.) Although what you read about in this notebook may, at first, seem relevant only to my other notebooks and OTTER, a bit of thought will show you that the ideas apply to automated reasoning in general and, for that matter, to reasoning itself. In other words, although here and in the other notebooks I focus on the use of OTTER, the approaches and strategies and the like are not dependent on OTTER.

---

*This material was based upon work supported in part by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357.

With OTTER, my colleagues and I have found proofs that had eluded fine minds for many, many years, proofs in group theory, equivalential calculus, and others. Rather often, millions and millions of conclusions were drawn before the desired proofs was found. (Indeed, in one of my studies reported in an earlier notebook, 10 million conclusions were drawn, of which more than 1 million were retained, before MCune's program completed its assignment.) When you obtain a result from OTTER, and many other reasoning programs, you can trust it explicitly, for, if you have not given the program erroneous information, no errors are made. Also, copious detail is typically provided, in contrast, say, to papers that you might be asked to referee.

In addition to complicated assignments of an arithmetic nature and assignments whose focus is data storage and retrieval (which you are most likely so familiar with), perhaps totally new to you is the existence of computer programs that often reason very effectively for a variety of applications. If you yourself write computer programs, you may know that large and fast reasoning programs exist whose purpose is to verify that some given computer program is free of bugs; that activity is called program verification. Some of those programs are called automated reasoning programs. The favorite automated reasoning program of mine, as noted, was written by William McCune and is called OTTER, and my use of it focuses on problems taken from diverse areas of mathematics and logic; OTTER is not used for program verification.

In this notebook, I provide you with a primer, a beginning, discussing the basics of automated reasoning in a manner that requires no background on your part. You will learn of many things: of a language that conveys the assignment to an automated reasoning program, of rules that enable such a program to draw conclusions that follow inevitably, of various types of strategy that markedly increase the effectiveness of such a program—and more. Because of the type of question I am sometimes asked about history—and perhaps to point you to browsing on the Internet—I shall touch on the vital role various colleagues have played in automated reasoning. Excitement is what I offer here.

You also will behold successes that, but a few years ago, appeared to be unreachable. The answer to a question that might immediately arise is, no, the breakthrough is not a result of bigger and faster computers; rather, the sharp increase in power rests with the results of research in automated reasoning. I shall tell you about programs that can and do solve puzzles, prove theorems in mathematics and logic, apply reasoning to various problems, and far more. In fact I almost immediately present a puzzle that was solved by an automated reasoning program, a puzzle for you to enjoy and solve.

You may naturally wonder how you can tell a program about such a puzzle or, for that matter, about deep theorems (to prove) taken from mathematics or logic. Of course, as you conjecture correctly, everyday language is not the means. You also may wonder how reasoning can be programmed, about procedures that enable conclusions to be drawn. At a deeper level, you may wonder how to ensure that the program doesn't get lost; after all, when trying to solve a puzzle or prove a theorem, getting lost is easy to do. At this deeper level, you may wonder about some approach that, possibly, permits you to give some guidance to the logical reasoning, some hints about how and where to go next. And what about types of redundancy? Drawing and then retaining the same conclusion perhaps a hundred times is far from appealing. In this notebook, your curiosity will be gratified. And, now for a puzzle.

2. The Jobs Puzzle

For the puzzle I now present, I quote and borrow (or steal) from a book I wrote (with others) many, many years ago.

> There are four people: Roberta, Thelma, Steve, and Pete.
> Among them, they hold eight different jobs.
> Each holds exactly two jobs.
> The jobs are chef, guard, nurse, clerk, police officer (gender not implied),
> teacher, actor, and boxer. The job of nurse is held by a male.
> The husband of the chef is the clerk.
> Roberta is not a boxer.
> Pete has no education past the ninth grade.
> Roberta, the chef, and the police officer went golfing together.

Question: Who holds which jobs?

This puzzle has been given to intelligent sixth graders, some of whom have solved it. If you cannot solve it, it may just prove that you are not an intelligent sixth grader, and nothing is wrong with not being classed as a sixth grader.

Of course, as you have surmised, I shall give you time to solve the puzzle before focusing on a method for obtaining the answers, a method a person might use. I then plan to solve this puzzle with you in a manner that an automated reasoning program can apply—focusing on an approach to finding the answers. (For that person who is eager to experience the use of OTTER, and especially in the context of the jobs puzzle, I shall present in Section 6 an appropriate input file.) In the meantime, some possibly subtle points need addressing.

Indeed, implicit information is present in the puzzle, information that a person must use to solve the puzzle and that a reasoning program must be given. As you and I take this journey into problem solving, you will begin to learn how to communicate with a program that reasons. Let us begin by examining the abundance of implicit information present in the puzzle's statement. You have most likely concluded that if the implicit information is not used (and not given to the automated reasoning program in use), then the program will not be able to solve the puzzle.

Especially if you have had little or no experience with computers, whether you are age 4 or 400, you might keep in mind that in general computers know nothing about people as well as other common items. For example, to enable your reasoning program to begin its attack on the jobs puzzle, you must convey to it the implicit information that Roberta and Thelma are female and that Steve and Pete are male. After all, you strongly suspect that such knowledge, as well as properties of being female or male, is crucial to solving the jobs puzzle. For various automated reasoning programs, including OTTER, the clause language is used. (Many reasoning programs today use the first-order predicate calculus; but since I plan for this notebook to accompany others I have on my website, automatedreasoning.net, I shall here focus on the clause language.) Its use would, most likely, have you write the following.

```
FEMALE(Roberta).
FEMALE(Thelma).
MALE(Steve).
MALE(Pete).
```

(Other possibilities exist.)

Of course, as you no doubt have concluded, other facts concerning female and male might be required, facts that a person knows but the program does not. For example, everybody is female or male, but not both. So, what might be a way, using the clause language, to impart this knowledge to an automated reasoning program? The following is just fine, where the " | " sign denotes logical **or**, and the "-" denotes logical **not**.

```
(1)  FEMALE(x) | MALE(x).
(2)  -FEMALE(x) | -MALE(x).
```

An explanation is immediately called for, especially for the second of the two statements just given. First of all, "x" is a variable meaning "for all"; for all x, x is female or male in (1), the first of the two given items. The second statement is a bit trickier. It can correctly be interpreted as "if x is female, then x is not male". Equally, it can correctly be interpreted as "if x is male, then x is not female". In other words, (implicitly) a person cannot be both female and male. (For the person who wishes to view the formal rule: For statements P and Q, the composite statement **if** P **then** Q translates to **not** P **or** Q.)

For an example of certainly-not-profound reasoning, you see that Roberta is not male, which (formally) you conclude from Roberta is female and item (2). The program, with a rule to be discussed, could draw this conclusion and retain it in the following form.

```
(3) -MALE(Roberta).
```

But you note, perhaps with a bit of impatience, the program has not yet made any progress toward finding

out which jobs Roberta has or, for that matter, which jobs the others have. So, let us quickly introduce a bit more and illustrate what a reasoning program might, and can, do.

You have another piece of implicit information, namely, that husbands are male. Do you see how this fact can be represented in the clause language in use here?

-HUSBAND(x,y) | MALE(x).

For any x and y, if x is the husband of y, then x is male. Well, Roberta is not male, for any x; therefore, Roberta is not the husband of anybody, of any y.

(4) -HUSBAND(Roberta,y).

So, coupled with the fact that the husband of the chef is the clerk—which can be rephrased as the person who holds the job of clerk is the husband of the chef—you see, and the program would with (4), that Roberta is not the clerk. From the program's viewpoint, the following might be in use.

(5) -HASAJOB(x,clerk) | HUSBAND(x,chef).

You, and your program, can obtain (6) from (4) and (5).

(6) -HASAJOB(Roberta,clerk).

Do you see how (6) was obtained? In particular, the program can use a rule of reasoning on (4) and (5) that has it, first, substitute in (4) chef for y, to obtain (4a), and also substitute in (5) Roberta for x, to obtain (5a). With the rule, called *ur-resolution* (formulated by Overbeek), the program takes (4a) and (5a) and so-to-speak cancels the second item (literal) of (5a) to obtain (6). And, at this point, you have eliminated, by applying logical reasoning, one of the possible jobs for Roberta, namely, the job of clerk. Of course, you do not yet know, nor does the program, who does hold the job of clerk.

As you seek the solution to this jobs puzzle, you might be using an approach taken by many puzzle-solvers. That approach consists of making a table whose four columns are labeled with the names of the four people and whose eight rows are labeled with the eight jobs. When you learn that, for example, Roberta is not the clerk, you place a "no" in the appropriate square, where the Roberta column intersects the clerk row. A program can, and in fact did years ago, (in effect) use this table approach. (Later, I shall show you how a program can apply this table approach.)

But although some detail has just been supplied with the discussion of (4a) and (5a), you have not yet been introduced, in its generality, to one of the rules that an automated reasoning program could use to proceed, as I just did, to deduce that Roberta is not the clerk. I hope you are somewhat eager to learn of such a rule.

The rule I have in mind is called *binary resolution*. McCune's program OTTER, as well as other automated reasoning programs, offers this rule, called an *inference* rule, that can be used to draw conclusion (6). Binary resolution always focuses on two items (nonempty clauses) and, if all requirements are met, draws a conclusion. The conclusion may be a unit clause, or it may be a nonunit, a clause in which **or** (denoted by " | ") is present. (The conclusion may also be the empty clause, to be implicitly interpreted as **false**.) Note that this is unlike ur-resolution, which requires that the conclusion be a unit clause. The requirements focus on needed substitutions of terms for variables—as illustrated with (4a) and (5a)—if variables are present, as well as other requirements.

To summarize, (6) can be deduced either with binary resolution or with ur-resolution. With either of the two cited inference rules, Roberta of (4) is substituted for the variable *x* in (5), the term chef of (5) is substituted for the variable *y* of (4), the (literal) or item of (4) is opposite in sign from the second term (literal) of (5), and the result is (6) after substituting Roberta for *x*. Conclusion (6) is a unit clause (or a unit), so-called because it is simple in the sense that it corresponds to some information in which logical **or** does not occur. The information can be positive or negative.

At this point, my guess says that some of you wish to see an example of a pair of statements that appear to yield a conclusion with either inference rule but that in fact do not because of failing some condition, perhaps that of an appropriate substitution. The following pair suffices.

EQUAL(x,x).

-EQUAL(y,s(y).

You can, loosely, interpret the function *s* as successor, 1 greater than the number in focus. As you have perhaps concluded, you cannot find two substitutions, one for the variable x and one for the variable y, that, when applied, cause the two resulting literals (items) focusing on equality to become identical, ignoring sign. In other words, no conclusion can be logically and correctly drawn from the two clauses. In particular, if appropriate substitutions could be found, the two given clauses would provide a contradiction.

Some brief observations about Ross Overbeek and William (Bill) McCune might provide you with an interlude while you digest the foregoing. I have known Ross since 1971, and, upon meeting him, was certain that he was brilliant. He was and is, although he might not agree. His programs of the 1970s, where the name automated reasoning had not yet been introduced—coined by me in I believe 1979—were remarkable. He and I are friends in the strongest interpretation of that term, and we still discuss the field often. In fact, it is Ross who is the wellspring for the existence of the various notebooks of mine that you find on my website. Bill, the author of OTTER, demonstrated the power of this program by using it to make many, many advances in mathematics and in logic, sometimes relying on another of his programs, EQP. He, as some of you may know, solved (with EQP) the long-standing open question concerning Robbins algebra—a fantastic achievement. He and I collaborated frequently, and I know of no more satisfying collaboration. Well, you have rested, had some wine or food, shared some of what has so far been written with one or more friends, and are ready for more.

You have briefly tasted the language, the clause language in particular, that can be used to communicate with an automated reasoning program. And you have had an even briefer taste of some of the methods used by the program to draw conclusions. Now you are wondering—I hope eagerly—about strategy. For example, you might ask why the program does not get lost in the myriad conclusions that it might draw. After all, I have only touched on some of the information you must give to ask your program to find the answers to the jobs puzzle. You might know, or have guessed, that one or more ways exist to restrict the program's search for relevant information. Eureka! There do exist strategies, called *restriction strategies*, to curtail the program's search, to decrease the likelihood of getting lost and wandering through a maze of uninteresting and useless conclusions. Indeed, since in the puzzle under study much information exists about people and their properties, if not restricted, a reasoning program might waste its time by drawing many, many conclusions of a general nature regarding people and jobs.

A strategy that is often most powerful is called the *set of support strategy* (which I formulated in the early 1960s), a strategy in which (if using OTTER) you place thought-to-be key items on a list called list(sos). With this strategy, the program is not allowed to consider for conclusion-drawing a set of items originally not on this list. The following illustrates good choices for items to place on that list.

FEMALE(Roberta).
FEMALE(Thelma).
MALE(Steve).
MALE(Pete).
-HASAJOB(Roberta,boxer).
-GREATERTHAN(education(Pete),9).
-HASAJOB(Roberta,chef).
-HASAJOB(Roberta,police).

The first four items are implicit, based on the knowledge of the gender associated with the names of the four people. The seventh and eighth items result from the (implicit) information that three people went golfing, one of who was Roberta, where it is implied that three *different* people went golfing. Do you see what is common to all of the eight items I would, and did, place on list(sos), thus causing the program to restrict its reasoning, initiating a line of reasoning only if the beginning of that line focuses on one of the eight items on list(sos)? Well, no variable occurs in any of the eight; each of them is specific, rather than general.

Now, in keeping with the thoughts of those who play chess, poker, and other games—not all of chance—and to complement restriction strategies, perhaps one or more *direction* strategies exists. Such a

direction strategy would guide the program toward profitable lines of reasoning and away from unprofitable lines of deduction. Indeed, with some programs, including OTTER, you can (in effect) give the program advice by including, for example, the following.

```
weight_list(pick_and_purge).
weight(Roberta,1).
weight(Thelma,2).
weight(Steve,3).
weight(Pete,4).
end_of_list.
```

With this inclusion, you are telling the program to give priorities to drawn-and-retained conclusions that focus on one of the four people. Further, you are instructing it to prefer, in order, retained information about Roberta, Thelma, Steve, and Pete. In other words, just as you might do, with the given instruction the program will focus hardest on Roberta (since it has more information about her), then on Thelma, and so on. For the curious about automated reasoning programs in general, (in my opinion) far too many of them do not offer a wide range of strategy. Indeed, a large and incredibly powerful computer does not suffice without strategy to aid you in finding answers and proofs in enough situations. Because of relying on a number of strategies, my colleagues and I have answered questions that had remained unanswered for many, many decades, answered with an automated reasoning program.

I plan to return to the jobs puzzle later in this notebook; but, as an essential part of this primer, other aspects of automated reasoning merit addressing.

3. More Features of Powerful Automated Reasoning Programs

At this point, I wonder whether you are ready to tackle some not-obvious tough questions. The questions focus on the fact that, in general, an automated reasoning program understands nothing. Indeed, that is why, to ask such a program to attempt to solve the jobs puzzle, you had to include knowledge that you have but the program does not, such as facts about everybody is female or male but not both. Have you thought of some notion, some relation, some property that is common to so many problems and questions, especially in areas of mathematics? Yes, you have chosen a good one: equality. Various reasoning programs throughout the decades have offered what might be termed a built-in understanding of equality. Further, many programs, including OTTER, offer a rule, an *inference rule*, for drawing conclusions justified by equality substitution. When in the late 1960s I formulated the rule in focus, called paramodulation, for quite a number of years it was not offered by all of the reasoning programs that occupied the attention of my colleagues. For the past few decades, however, OTTER and many of its predecessors (designed and implemented by Overbeek) and other programs have offered the user paramodulation when equality is in play. To clarify, McCune designed and implemented OTTER in 1988 (if memory serves), and paramodulation played immediately a prominent role.

A look forward, in the context of paramodulation, seems in order. In various types of problem solving, certainly in many areas of mathematics, equality plays a dominant role in the reasoning required to reach the desired goal. At the simplest level, one commonly encounters the practice of substituting equals for equals. Paramodulation—and here comes the observation to intrigue you—generalizes this property of substitution, and in a manner that even mathematicians would not wish to apply unaided but that a reasoning program such as OTTER finds easy to apply. In Section 4, you will be treated, in great detail, to the complexities and subtleties of paramodulation, and you will learn how equality substitution is generalized by the use of paramodulation.

At this point, with more details to come in Section 4, you might find pleasant the following natural example of the way paramodulation works. For the example, allowing myself to use equality in a less precise manner, I offer the observation that Overbeek is the friend of Wos and the observation that the friend of Wos is smart, and I ask the program using paramodulation to draw a conclusion.

```
EQUAL(friend(wos),Overbeek).
IS(friend(Wos),smart).
IS(Overbeek,smart).
```

The third of these three items is a conclusion justified by paramodulation: Overbeek is smart. When a user instructs a reasoning program to rely fully or in part on paramodulation, an easily overlooked item to be input is the clause x = x, which is a property known (to so many of you) as reflexivity. After all, perhaps surprisingly often, as the program draws one conclusion after another, among such is one of the form t != t for some term t. If $x = x$ is not present, then the obvious contradiction is not recognized.

Oh, I did not point out that many assignments, but certainly not all, (in effect) ask the reasoning program to eventually establish a contradiction between two unit statements, one of which is often among the input items. In particular, when seeking a proof in some area of algebra, geometry, or logic, a proof by contradiction typically is sought. Certainly, in my years of research, almost without exception, I have asked a reasoning program to find a proof by contradiction, as occurs so often in a (university) class in mathematics or logic. So, if you decide to experiment with some automated reasoning program, and if you choose to rely on paramodulation, you would be wise to include in the input reflexivity of equality, $x = x$. Indeed, perhaps surprisingly often, a reasoning program deduces something like a != a, for some constant a. If such a deduction occurs and is retained, a contradiction is detected with the axiom of reflexivity.

You have now had a small taste of one of the implied questions to tackle, the automated treatment of equality in an efficient manner, and have perhaps thought of another. Specifically, as is true when trying to solve a puzzle or answer a question by hand, seldom is the path to success unique. In fact, I wager that you have on more than one occasion reasoned to some conclusion two, three, or more times. You have found a number of paths that start with a set of items and ended with the same conclusion, and not a conclusion that is the goal. I am, of course, talking about a type of redundancy. Typically, when you arrive at a point for a second time, or worse, you discard the corresponding conclusion. One so-to-speak copy of the deduction suffices. Well, for needed power and efficiency, an automated reasoning typically offers a procedure, called *subsumption*, that discards duplicate information. (I believe Alan Robinson merits the credit for introducing subsumption to automated reasoning.) Even better, as you will immediately see, subsumption is used to discard so-called trivial instances of an already-retained item. For example, in everyday language, if the reasoning program deduces that wives are female and later deduces that Steve's wife is female, the latter deduction is immediately discarded as being an instance of the former.

FEMALE(wife(x).
FEMALE(wife(Steve).

In so many, many problems whose solution is sought with the aid of an automated reasoning program, the instruction to continually apply subsumption is virtually required. (I often think of and refer to such a program as an automated reasoning assistant.) For a slightly more interesting example, if such a program has deduced that all wives are female, the deduction that all wives are female or male is discarded by subsumption. You might be amused to learn that, in some of the problems (from areas of mathematics and logic) I have considered with the aid of a reasoning program, the use of subsumption has resulted in the discarding of millions of deduced conclusions. The time required to, say, find the desired proof was ordinarily sharply reduced with the use of subsumption.

Now, with this discussion of a type of redundancy addressed with the use of subsumption, you may have thought of another question to consider, also focusing on a type of redundancy. In particular, if you are having the reasoning program consider a problem from arithmetic and if the program deduces that $a+b = c$ for constants a, b, and c from the problem, the deduction of $a+b = c+0$ is not very inspiring. Worse, in the obvious sense, is a replacement of c in a later deduction by $0+c+0+0$. (Of course, you can construct a bizarre case in which this last deduction is useful.) Those of you who are mathematicians know, for example, that in group theory $0+x = x$ in the beginning. The deduction of $0+0+0+x+0 = x$ does not produce awe. Yes, in arithmetic and group theory, respectively, the given examples can and do occur. So, for efficiency, you might ask about some aspect of automated reasoning that addresses this type of redundancy. Further, for variables x, y, z, and u, with associativity, you can deduce $((x+y)+z)u = (x+y)(+z+u)$, as other similar expressions with different use of parentheses. So, again, does there exist a commonly available procedure in a good reasoning program to address the attendant problems?

The answer is yes. Various programs, among them OTTER, do in fact offer just what is needed, namely, a procedure called *demodulation*. If, for example, you wish to rewrite deductions that involve 0

(and are redundant in the sense under discussion) and those focusing on various associative ways of expressing an item, for OTTER you would include the following.

```
list(demodulators).
EQUAL(0+x,x).
EQUAL(x+0,x).
EQUAL((x+y)+z,x+(y+z)).
end_of_list.
```

The use of demodulation in areas such as group theory sharply increases the efficiency of a reasoning program in the vast majority of cases. (I note that I formulated demodulation in the late 1960s as a result of studying a single failed attempt at finding a proof.) Further, in that example, rather than a sharp reduction occurring from rewriting various deduced items, the sharp increase in efficiency resulted from the absence of the so-called redundant items that would have been used to spawn children, then grandchildren, and so on.

In other words, sometimes the use of demodulation does not, at the first level, cause the program to rewrite a large number of deduced items; instead, its use cuts off a few paths that would produce (at higher levels) many, many items. Although demodulation is ordinarily relied on in assignments involving equality, it does have other uses. In various areas of logic, in particular, I have profitably used it to purge any deduction that relied on what is called double negation. For example, if the program deduced a formula containing the term $n(n(x))$, where the function n denotes negation, I would have OTTER discard it. Steve Winker, many years ago, showed how demodulation—typically used for simplification and canonicalization—could be used in a number of (on the surface) most unusual ways.

At this point, you have tasted the representation of information to an automated reasoning program, viewed some of the inference rules it can use to draw conclusions on the way to completing an assignment, and witnessed a bit of strategy to restrict and a bit to direct its reasoning. I also hinted that typically a proof by contradiction is sought as the goal in mathematics and in logic. A bit more is merited now concerning this concept of proof by contradiction.

For the first example, a simple one indeed, if the puzzle only asks for you to prove that Roberta is a teacher, then typically you would include in the presentation of the puzzle the assumption that Roberta is *not* the teacher. Then, if, as is the case in the jobs puzzle, your reasoning program deduces along the way that Roberta has the job of teacher, that deduction together with the assumed (and false) item that Roberta is not the teacher enables the program to stop its search with success. Pedantically, the program has found a contradiction between Roberta is a teacher and Roberta is not a teacher. A second example is clearly in order.

For this example, do not be misled by its simplicity, for I shall return to it later with a challenge for you. In arithmetic, operations such as addition and multiplication are known to be commutative. Indeed, if you multiply two expressions in either order, you obtain the same result. Other areas exist for which commutativity may not hold or for which it can be proved from the properties of the area in focus. So, imagine you are reading of such an area and are given some properties with the claim that commutativity can be proved to hold. You are asked to prove commutativity and, therefore, tell your reasoning program that $ab \mathrel{!}= ba$, for some a and some b; you assume that the result is not true. If the program eventually, or quickly, deduces $xy = yx$, for all (variables) x and y, then it can signal that a proof by contradiction has been completed, using the given (false) assumption.

And now for the challenge, especially for the person who enjoys some mathematics. For the area of mathematics called group theory, various professors offer their students early in the course the following theorem to prove. You are told that a group is a nonempty set of elements in which multiplication exists (for every pair of elements, a third element exists that is their product). You are also told that, with regard to multiplication, associativity is present, $(xy)z = x(yz)$. And you are told that there exists an identity element e with $ex = xe = x$, and, with respect to e, an inverse exists. In other words, for every x, there exists a y such that $xy = yx = e$. You now know what a group is, and you realize that the property of commutativity, $xy = yx$, is not assumed to be present. Well, if you now assume that, for all x, $xx = e$, then you can prove that the group is commutative, that $xy = yx$. And that is your challenge: Prove that groups in which $xx = e$

are commutative.

A program such as OTTER can, as you have immediately guessed, find the desired proof. If you had a copy of OTTER, or some other reasoning program with some power, you could give it the properties (axioms) of a group and the additional property that *xx = e* and ask it to prove commutativity. You would also, to seek a proof by contradiction, include *ab != ba*, for some (constants) *a* and *b*. Depending on which lists you placed which items, at least for OTTER the program might deduce *xy = yx*, which would signal the completion of a proof by contradiction when taken together with *ab !=ba*. Or, the program might reason backward from *ab != ba* and forward from the axioms and the additional assumption that *xx = e* and meet somewhere in the middle. Or the program might simply reason backward until a contradiction was found. Just for illustration, the program might deduce (*ab*)(*ab*) != (*ba*)(*ab*), then (eventually) (*ab*)(*ab*) != *e*, by applying associativity and the *xx = e* property. If this deduction were made, it would contradict *xx = e*, and success would be your program's—and yours.

At this point, you may well be having difficulty choosing from three possibilities: the prospect of viewing more strategies, of learning how a reasoning program might function as you might when using a table approach to the jobs puzzle, and of learning of some of the problems still focusing on automated reasoning. A wiser choice might be obvious; but lacking the needed evaluation, I shall now turn to the third of the three choices.

4. Obstacles to Be Overcome

If you are not ready for more complex issues, you might skip this section. On the other hand, in the spirit of computer science and mathematics and logic, you might thirst for some depth. Here, I pose two questions for automated reasoning for which I do not at this time have answers. The first question focuses on "case analysis". For example, when you are trying to solve a puzzle, you might consider two paths: Kim is female, and Kim is male, each as a separate case, a separate subpuzzle. After all, the name Kim (as I treated in a book I wrote years and years ago) does not provide a clue concerning gender. For the first case in the case analysis, you would make the assumption that Kim is female and pursue the resulting path to a solution. For the second case, you would assume that Kim is male, pursuing the resulting path to a solution. If both paths lead to the desired answer to the puzzle, since Kim must be female or male, and none other, you have solved the puzzle.

For an example pertinent to the type of research I do, in a problem or theorem from mathematics, you can easily see how case analysis might come into play, namely, *a = b* and *a != b* for two constants *a* and *b* in the problem, of course, where equality is playing a role. You yourself, or your reasoning program, could assume, first, that *a = b* and seek a proof with this added assumption. If a proof is found, you, or your program, could then pursue the case *a != b* and seek a corresponding proof. Since in the example under discussion only the two cited cases exist, you would have found the answer, or the proof, with case analysis. You might immediately think of a third example: *c*, an element, is in the subset *T*, and *c* is not in *T*. Of course, some purists would ask for a reasoning program to avoid case analysis and simply seek the desired answer or desired proof without an added assumption corresponding to one of a set of cases suggested by case analysis.

In the late 1960s, if memory serves, I was asking the program then in use—when the field was called automated theorem proving—to prove a theorem from group theory. The theorem asserts that subgroups of index 2 are normal; in other words, you have a group *G*, and you have a subgroup *H* of index 2 in *G*. If you lack the background for understanding what is to be proved, fear not; indeed, I am about to discuss an approach that sometimes succeeds in place of case analysis. In that study, I was able to obtain from the program in use a proof by seeking two proofs, one for each of two cases of the type cited in the third example; one case focuses on some chosen element, *b*, in the subgroup *H* (of index 2), and the other focuses on the case where *b* is not in *H*. Now, with the purist in mind, I note that, without breaking the problem into two subproblems, I could not prove the theorem with the program then in use. However, I decided on another approach, one that differs from using case analysis. When equality is in play, rather than two cases, of the = and != type for example, you can sometimes adjoin an appropriate tautology, namely, *a = b* **or** *a != b*. The tautology I used for this problem, focusing on subgroups of index 2, was of the following form: *c* is

in the subset *T* **or** *c* is not in the subset *T*. And success resulted. In other words, two separate subproblems were replaced by a single run seeking a proof, a run in which a tautology was adjoined. The question for you—the challenge—is first, under which conditions will success occur with the adjoining of a tautology? More important, why does adjoining a tautology work and when? After all, a tautology of the type just cited is always **true**, so why should its presence facilitate finding an answer or a proof?

For a second challenge, quite different from the preceding, the focus is on the inference rule paramodulation, a rule for drawing conclusions when equality is in focus. I warn you that you are about to enter a jungle, for to understand paramodulation may require much patience, as the following three examples illustrate. These examples also show you why paramodulation has proved virtually indispensable for much of success that has occurred in the past forty years and more. The third example also will most likely show you why a person would almost never apply this inference rule by hand.

For the first example of paramodulation, I offer you three clauses, the third of which, the conclusion obtained from the first two, follows from so-to-speak straightforward reasoning.

EQUAL(sum(a,minus(a)),0).
CONGRUENT(sum(a,minus(a)),b).
CONGRUENT(0,b).

An automated reasoning program captures this straightforward bit of reasoning by using paramodulation, *from* the first clause *into* the second clause, to obtain the third. Indeed, the example illustrates the usual notion of equality substitution.

For the second example, equality-oriented reasoning applied to both the equation $a + (-a) = 0$ and the statement "$x + (-a)$ is congruent to x" yields in a single step the conclusion or statement "0 is congruent to a".

EQUAL(sum(a,minus(a)),0).
CONGRUENT(sum(x,minus(a)),x).
CONGRUENT(0,a).

Again, paramodulation suffices, reasoning *from* the first of the following three clauses, *into* the second, obtaining the third. This example illustrates some of the complexity of the use of equality and of paramodulation. In particular, the second occurrence of the variable x in the *into clause* (second clause) becomes the constant a in the conclusion (third clause), but the (larger) term containing the first occurrence of x becomes the constant 0 in the conclusion. A bit of thought shows that the left-hand argument of the equation (first clause) can be directly applied to the left-hand argument of the congruency statement (second clause) if one merely sets the variable x to the constant a in both clauses. For a program such as OTTER, this all happens automatically through the use of *unification* when the focus is on the two left-hand arguments. 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 (in the second clause) to be replaced by the constant a, paramodulation then requires an additional term replacement justified by straightforward equality substitution, the replacement of $a + (-a)$ by 0.

To show why paramodulation is not the type of reasoning a person would ordinarily apply by hand, I offer a third example, one that some people versed in mathematics found hard to accept. Indeed, the example illustrates a perhaps unexpected subtlety—equality-oriented reasoning applied to both the equation $x + (-x) = 0$ and the equation $y + (-y + z) = z$ yields in a single step the conclusion $y + 0 = -(-y)$, a conclusion that might at first seem to be unsound. Let us consider the example in clause form.

EQUAL(sum(x,minus(x)),0).
EQUAL(sum(y,sum(minus(y),z)),z).
EQUAL(sum(y,0),minus(minus(y))).

Paramodulation suffices here as well as earlier, applied to the following three clauses, *from* the first *into* the second, to logically deduce the third (again without producing any intermediate clauses). Of course the conclusion corresponds to a well-recognized truth. But how does it follow from flawless reasoning applied to the two hypotheses?

To see that this last clause is in fact a logical consequence of its two parents, you unify (with the appropriate substitution of terms for variables) the argument sum(*x*,minus(*x*)) with the term sum(minus(*y*),*z*), apply the corresponding substitution to both the *from* and *into clauses*, and then make 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 minus(*y*) for *x* and minus(minus(*y*)) for *z*. In order to prepare for the (standard) use of equality in this third example—and here you encounter a key feature of paramodulation—a nontrivial substitution for variables in both the *from* and the *into clauses* is required, which illustrates how paramodulation generalizes the usual notion of equality substitution. In contrast, in the standard use of equality substitution, a nontrivial replacement for variables in both the *from* and the *into* statements is not encountered.

Summarizing, a successful use of paramodulation combines in a single step the process of finding the (in an obvious sense) most general common domain (through unification) for which both the *from* and *into clauses* are relevant and applying standard equality substitution to that common domain. The complexity of this inference rule rests in part with its unnaturalness (if viewed from the type of reasoning people employ), in part with the fact that the rule is permitted to apply nontrivial variable replacement to both of the statements under consideration, and in part with 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 given, the (term containing the) first occurrence of *z* is transformed to 0, but the second occurrence of *z* is transformed to minus(minus(*y*)). The third example illustrates what I mean when I say that, rather than literal-oriented (as is the case for hyperresolution), paramodulation is a term-oriented inference rule that generalizes equality substitution. (Fom the individual who finds history entertaining, and even amusing, I note that when I first formulated paramodulation, I failed to consider its use for nonunit clauses; but, thanks to George Robinson, my colleague at the time, the use of paramodulation in the context of nonunit clauses was included.)

Since equality works the way it does, given an equality and another expression, the two may be such that, from the pair, you can deduce many logical and correct conclusions with this term-oriented inference rule paramodulation. After all, since you are allowed with equality to substitute equals for equals, the so-called *into* expression may have many, many subexpressions that you can substitute for. Clearly, paramodulation can be a prolific inference rule. With its use, gold has been mined in various areas of mathematics. For but one example, paramodulation played a key role in McCune's success in answering the long-standing open question that asked whether the axioms for a Robbins algebra imply that every Robbins algebra is a Boolean algebra. He did, in fact, prove that every Robins algebra is a Boolean algebra, in part benefiting from much earlier work of Steve Winker.

I now come to a difficult problem to solve, a substantial challenge, even if you are well-schooled in automated reasoning. How can the use of paramodulation be controlled so that its use produces far fewer conclusions than its use can, under many situations, produce? Two strategies do exist for modulating the prolificity of paramodulation. The first has the program *never* paramodulate *into* a term *t*, where *t* is a variable. The second strategy has the program avoid paramodulating *from* a variable. For an example of the second strategy, you can consider the following equality.

EQUALsum(x,0),x).

The given equality is, of course, familiar in arithmetic and, for mathematicians, familiar in, for example, group theory. With the restriction of not allowing paramodulation *from* a variable, the program would never consider trying to match (unify) the right-hand arugment, the isolated variable *x*, with any term of the *into* expression. The two given strategies are almost always relied on when paramodulation is in use. Additional powerful restriction strategies for the use of paramodulation would be met with substantial excitement.

5. Another Inference Rule, Another Strategy, and Another Approach

For this section, I begin with an approach that, on the surface, is novel—and its use has unearthed real treasure. This approach contrasts nicely with the typical approach of presenting to the reasoning program the facts, relations, and properties of the puzzle or theorem in focus. The approach, called *sketches*,

was formulated by Robert Veroff. Loosely speaking, what he does with sketches is the following. When his version of OTTER does not yield a proof of a given theorem under study, he adjoins a (possibly) large number of additional properties, many of which he is certain are not appropriate or valid in the area in focus. If and when he obtains a proof with the augmented input file, he gradually removes the added assumptions to iterate to get one proof after another. The succeeding proofs rely on fewer and fewer adjoined items. From each proof, he extracts proof steps to use in the next run, used as hints for how the program should proceed in its pursuance of a proof.

Now, switching from an approach to an inference rule for drawing conclusions, I first remark that the deduction of unit clauses has always been to me key to solving problems. Also, their presence in the original problem or theorem presentation has always been welcome. Of a related nature, positive information is ordinarily more useful than negative. For example, the fact that something is not true or that two items are not equal abounds in nature, in mathematics and logic, and elsewhere. Therefore, a rule that promotes the deduction of positive information, unit or nonunit, is most appealing. *Hyperresolution* is such an inference rule. The object of an application of *hyperresolution* is to produce a positive clause from a set of clauses one of which (the nucleus) is negative or mixed while the remaining (the satellites) are positive clauses. The number of satellites must be equal to the number of negative literals in the negative or mixed clause, and the inference rule requires the result of successful application to be a positive clause. For an example, consider the following set of clauses.

-FEMALE(Roberta) | -HASAJOB(Steve,nurse) | MARRIED(Thelma).
FEMALE(Roberta).
HASAJOB(Steve,nurse).

If hyperresolution were applied to the three clauses all at once, the program would conclude that Thelma is married. For a successful application of hyperresolution, the program must deduce a positive clause, including the empty clause consisting of no literals, but the deduced conclusion need not be a unit clause.

Completing the subjects of this section title (though not in the order suggested), I next introduce another strategy, called the *hot list strategy*. As background and motivation for this strategy, formulated by me in the early 1980s, I note that when you are attempting to solve a hard puzzle or prove a deep theorem, you (almost for certain) visit and revisit one of the items presenting the puzzle or theorem. For example, with the jobs puzzle, you might visit and revisit items about Roberta, even when trying to find out which jobs Thelma has. For an example taken from mathematics, if the theorem comes from ring theory and you are told that $xxx = x$ for all x, you likely would repeatedly key on this added assumption when trying to prove commutativity. In other words, one or more items that present the puzzle or theorem are typically emphasized as you search for the desired answer.

Well, the hot list strategy offers just what is needed, and more. With this strategy, you place one or more items in a hot list; each time the program decides to retain a new conclusion, it attempts to use each of the items on the hot list with the new conclusion to draw additional conclusions. Moreover, the hot list strategy offers additional power, depending on a value you assign to the appropriate parameter. At least, that is how OTTER offers the hot list strategy. Indeed, you can have the program, when a new conclusion is retained, try to deduce children whose parents are the new result and a member of the hot list, and then immediately try to deduce grandchildren, great grandchildren, and so on. Moreover, thanks to McCune, you can choose appropriate instructions to have your program (with the hot list strategy) adjoin dynamically new items to the (input) hot list during the attempt to reach the desired goal. In other words, sometimes you can instruct your program to perform in a manner that emulates that which a person takes, but extend this emulation.

But you virtually demand at this point, after so much material, to have an input file that you can run with OTTER—and a discussion of the paths of reasoning that culminate in the solution to the jobs puzzle.

6. Input File and Paths of Reasoning Used to Solve the Jobs Puzzle with OTTER

If you take the input file I am about to supply, and if you have a copy of OTTER or download one by consulting my website automatedreasoning.net, you can make a gratifying experiment. In particular, you can find out who holds which jobs. And even if you have already deduced the answer, you may find

amusing that a reasoning program can also solve the puzzle, and in a manner that is reminiscent of using a table. As a word of warning, note that in the following sets of clauses you will find uses of equality—with equality of people distinguished from equality of jobs to avoid drawing unsound conclusions. And as a word of encouragement, do not be disheartened by a first glance at the file: I promise to give a detailed explanation of the file shortly.

Input File for the Jobs Puzzle

```
set(ur_res).
set(back_demod).
% set(dynamic_demod_all).
set(input_sos_first).
clear(print_kept).
assign(max_proofs,1).

weight_list(pick_and_purge).
weight(Roberta,1).
weight(Thelma,2).
weight(Steve,3).
weight(Pete,4).
end_of_list.

list(usable).
FEMALE(x) | MALE(x).
-FEMALE(x) | -MALE(x).
-HASAJOB(x,nurse) | MALE(x).
-HASAJOB(x,actor) | MALE(x).
HASAJOB(x,job1(x)).
HASAJOB(x,job2(x)).
HASAJOB(jobholder(y),y).
-HUSBAND(x,jobholder(chef)) | HASAJOB(x,clerk).
-HASAJOB(x,clerk) | HUSBAND(x,jobholder(chef)).
FEMALE(jobholder(chef)).
-HUSBAND(x,y) | MALE(x).
-HUSBAND(x,y) | FEMALE(y).
-HASAJOB(x,nurse) | GREATERTHAN(education(x),9).
-HASAJOB(x,police) | GREATERTHAN(education(x),9).
-HASAJOB(x,teacher) | GREATERTHAN(education(x),9).
-HASAJOB(x,chef) | -HASAJOB(x,police).
-EQUALP(Roberta,Thelma).
-EQUALP(Roberta,Steve).
-EQUALP(Roberta,Pete).
-EQUALP(Thelma,Steve).
-EQUALP(Thelma,Pete).
-EQUALP(Pete,Steve).
-EQUALJ(job1(x),job2(x)).
EQUALP(x,x).
EQUALJ(x,x).
EQUAL(x,x).
EQUALJ(y,job2(x)) | EQUALJ(y,job1(x)).
EQUALP(x,z) | -HASAJOB(x,y) | -HASAJOB(z,y).
-FEMALE(jobholder(y)) | HASAJOB(Roberta,y) | HASAJOB(Thelma,y).
-MALE(jobholder(y)) | HASAJOB(Steve,y) | HASAJOB(Pete,y).
```

```
POSSJOBS(l(pj(Roberta,chef),l(pj(Roberta,guard),
  l(pj(Roberta,nurse),l(pj(Roberta,clerk),
  l(pj(Roberta,police),l(pj(Roberta,teacher),
  l(pj(Roberta,actor),l(pj(Roberta,boxer),end)))))))).
POSSJOBS(l(pj(Thelma,chef),l(pj(Thelma,guard),
  l(pj(Thelma,nurse),l(pj(Thelma,clerk),
  l(pj(Thelma,police),l(pj(Thelma,teacher),
  l(pj(Thelma,actor),l(pj(Thelma,boxer),end)))))))).
POSSJOBS(l(pj(Steve,chef),l(pj(Steve,guard),
  l(pj(Steve,nurse),l(pj(Steve,clerk),
  l(pj(Steve,police),l(pj(Steve,teacher),
  l(pj(Steve,actor),l(pj(Steve,boxer),end)))))))).
POSSJOBS(l(pj(Pete,chef),l(pj(Pete,guard),
  l(pj(Pete,nurse),l(pj(Pete,clerk),
  l(pj(Pete,police),l(pj(Pete,teacher),
  l(pj(Pete,actor),l(pj(Pete,boxer),end)))))))).
POSSPPL(l(pj(Roberta,chef),l(pj(Steve,chef),
  l(pj(Thelma,chef),l(pj(Pete,chef),end))))).
POSSPPL(l(pj(Roberta,guard),l(pj(Steve,guard),
  l(pj(Thelma,guard),l(pj(Pete,guard),end))))).
POSSPPL(l(pj(Roberta,nurse),l(pj(Steve,nurse),
  l(pj(Thelma,nurse),l(pj(Pete,nurse),end))))).
POSSPPL(l(pj(Roberta,clerk),l(pj(Steve,clerk),
  l(pj(Thelma,clerk),l(pj(Pete,clerk),end))))).
POSSPPL(l(pj(Roberta,police),l(pj(Steve,police),
  l(pj(Thelma,police),l(pj(Pete,police),end))))).
POSSPPL(l(pj(Roberta,teacher),l(pj(Steve,teacher),
  l(pj(Thelma,teacher),l(pj(Pete,teacher),end))))).
POSSPPL(l(pj(Roberta,actor),l(pj(Steve,actor),
  l(pj(Thelma,actor),l(pj(Pete,actor),end))))).
POSSPPL(l(pj(Roberta,boxer),l(pj(Steve,boxer),
  l(pj(Thelma,boxer),l(pj(Pete,boxer),end))))).
HASAJOB(x,y) | EQUAL(pj(x,y),crossed).
EQUAL(l(crossed,x),x).
-POSSJOBS(l(pj(x,y),l(pj(x,z),end))) | EQUALP(x,w)
 | EQUAL(pj(w,y),crossed).
-POSSJOBS(l(pj(x,y),l(pj(x,z),end))) | EQUALP(x,w)
 | EQUAL(pj(w,z),crossed).
-POSSJOBS(l(pj(x,y),l(pj(x,z),end))) | HASAJOB(x,y).
-POSSJOBS(l(pj(x,y),l(pj(x,z),end))) | HASAJOB(x,z).
-POSSPPL(l(pj(x,y),end)) | HASAJOB(x,y).
STILLTODO(l(jobsof(Roberta),l(jobsof(Steve),
l(jobsof(Thelma),l(jobsof(Pete),end))))).
-POSSJOBS(l(pj(x,y),l(pj(x,z),end))) | EQUAL(jobsof(x),crossed).
-STILLTODO(end).
-HASAJOB(x,y) | EQUAL(pj(x,y),j(x,y)).
EQUAL(l(pj(x,y),l(j(x,z),w)),l(j(x,z),l(pj(x,y),w))).
EQUAL(l(j(x,y),l(j(x,z),l(v,w))),l(j(x,y),l(j(x,z),end))).
-POSSJOBS(l(j(x,y),l(j(x,z),end))) | EQUAL(jobsof(x),crossed).
-POSSJOBS(l(j(x,y),l(pj(x,z),end))) | HASAJOB(x,z).
end_of_list.

list(sos).
```

```
FEMALE(Roberta).
FEMALE(Thelma).
MALE(Steve).
MALE(Pete).
-HASAJOB(Roberta,boxer).
-GREATERTHAN(education(Pete),9).
-HASAJOB(Roberta,chef).
-HASAJOB(Roberta,police).
end_of_list.

list(demodulators).
EQUAL(l(crossed,x),x).
end_of_list.
```

Again I pause here to enable you to glean, if you choose to do so, more hints about solving the puzzle by hand, where the hints are provided by an examination of the given input file. I'll tell you what: If you wish to continue your approach to solving the jobs puzzle on your own—perhaps benefiting from a glance at the given input file—I shall ramble a bit, with some more history and other remarks, before summarizing what will be found or what can be found by a person unaided.

The first automated reasoning program produced at Argonne National Laboratory in 1963. It was designed and implemented by a fine assembly language programmer, Dan Carson. The important aspect, from my viewpoint, was the program's use of strategy, a strategy that placed an emphasis on unit clauses, nonempty clauses free of the **or** sign. The strategy is called the unit preference strategy. With the first version of Carson's program, little of interest was proved. But after the set of support strategy was added to the program, the next version proved a simple theorem from group theory, namely, if $xx = e$ in a group, the group can be proved to be commutative. (In the early 1960s, the field was called, not wisely I fear, mechanical theorem proving; later, it was called automatic theorem proving, which, of course, it was not; then it was called automated theorem proving; and finally—I believe in late 1979—the term automated reasoning was coined by me.) Our first serious research focused on group theory probably because I had received my Ph.D. in group theory under that great mathematician R. Baer. If you find interesting the birth of ideas, seek research problems to test the value of new enhancements, and would enjoy an anecdote that focuses on the set of support strategy and Carson, I suggest a book I wrote in the mid-1980s, *Automated Reasoning: 33 Basic Research Problems*, Section 2.1.1.

Well, it is time to present the answers to the jobs puzzle and to discuss the paths of reasoning that could be used.

```
HASAJOB(Roberta,teacher).
HASAJOB(Roberta,guard).
HASAJOB(Thelma,chef).
HASAJOB(Thelma,boxer).
HASAJOB(Steve,nurse).
HASAJOB(Steve,police).
HASAJOB(Pete,clerk).
HASAJOB(Pete,actor).
```

You may find it profitable to see how the reasoning that leads to the puzzle's solution might proceed. You could, for example, compare the reasoning—by person or persons unknown—with your own. As the exposition proceeds—taken from a book I wrote many, many years ago but tailored to this notebook—you will find observations made earlier in this narrative. The table approach cited earlier will be used.

A reasonable strategy a person might use is first to concentrate on Roberta's two possible jobs, for more is known—at least implicitly—about Roberta than is known about anyone else in the jobs puzzle. For a look-ahead, note that the program tries to concentrate on Roberta as indicated by the assignment of the value 1 to Roberta in weight_list(pick_and_purge). Roberta is not the boxer—that fact is given in the

puzzle. To learn more about Roberta and the jobs she holds, you must make some assumptions that are based on common usage of everyday language. The puzzle says that the nurse is male. Since Roberta is female—a fact implicit in her name—you may conclude that she is not the nurse. (The puzzle is subtly designed, for if the four names did not clearly imply the gender of the people, it would be impossible to solve.) You also know that she is not the actor. Why? Because everyday language distinguishes members of this profession based on gender—actor and actress. (Notice, however, that you cannot assume that the police officer is male, for the job is labeled police officer and not policeman.)

The puzzle says that Roberta, the chef, and the police officer went golfing together. This statement is rich in "hidden" facts, for in normal usage it means that three distinct people went golfing. Thus, you know that Roberta is neither the chef nor the police officer.

The husband of the chef is the clerk. Coupled with the implicit fact that husbands are male, this fact tells us that Roberta is not the clerk. This leaves the jobs of guard and teacher for Roberta. The other six jobs have been crossed off of her list—"no" can be placed in the corresponding squares in her column.

Notice that even though the reasoning that Roberta is not the clerk seems explicit—perhaps even painfully so—another bit of implicit information has been used in reaching that conclusion. The fact that females are not males also must be given to a reasoning program.

More squares can immediately be filled in. The puzzle says that there are four people and eight jobs and that each person holds exactly two jobs. Implicit in these facts, as remarked earlier, is the fact that no job is held by two people. Thus the jobs of guard and teacher can be crossed off the list of possible jobs for each of Thelma, Steve, and Pete. Doing so leaves six possible jobs for the three remaining people—eighteen squares still to be filled in—and everything is fine. The check that there are enough jobs left for the remaining people in the puzzle is an example of a useful process. Such checks are valuable in puzzle solving, whether the solving is done by a person or by a program.

With Roberta's jobs determined, the next person on whom to focus is Thelma. The choice of Thelma is consistent with the strategy that first pointed to Roberta. Again, as a look-ahead, with the assignment of the value 2, in weight_list(pick_and_purge), to Thelma, you see how the program is directed to next focus on Thelma. Although it might appear that you know no more about Thelma than you do about Pete, you, and the program, can use the items learned about Roberta for a similar argument about Thelma. You might assume that you know as much about Thelma as you do about Pete; that assumption is false. Implicit in her name is the fact that she is female. Some of the arguments used to determine Roberta's jobs therefore apply to Thelma. Thelma cannot be the nurse, the actor, or the clerk—those jobs are held by a male. Moreover, since the puzzle says that the chef has a husband, since husbands are male, and since Thelma is the only female left, she is the chef.

Since they went golfing together, the chef and the police officer are not the same person. So Thelma is not the police officer. The jobs of guard and teacher have already been crossed off her list. So she is the chef and—surprise—the boxer. (Nowhere was it implied that the boxer is male.)

The jobs for Steve and Pete can be quickly determined now. The puzzle says that Pete has no education past the ninth grade. To use this information, you must employ some deeply hidden assumptions—some that depend on our knowledge of jobs in our culture. In the United States, at this time (in 2014), the jobs of nurse, police officer, and teacher each require more than a ninth-grade education. Thus, Pete cannot be the nurse or the police officer. So Pete is the actor and the clerk. These jobs can be crossed off the list for Steve. Then the remaining two jobs, nurse and police officer, must be held by Steve.

This puzzle, like so many, is much more easily solved if a good strategy is used. The strategy used here is to concentrate on the person about whom you know the most but for whom the jobs are not yet determined. The mechanism that can be used to do so is known as *weighting*, a strategy due to Overbeek.

Another feature of this solution to the puzzle is interesting. Not only did you learn who held which jobs, but you learned also that Pete and Thelma were married. Finding such extra information is common in people's attempts at solving puzzles, and it is also common in a reasoning program's attempts. If too much extra information is found, the puzzle never gets solved—by a person or by a program. Thus, one of the important items you need to learn about in order to use a reasoning program effectively is how to curtail

the finding of extra information. Providing various ways to reduce exploration of fruitless paths is one of the powerful features of an automated reasoning program.

A few words, as promised, are now in order to enable you to understand the given input file. Perhaps the first thing you notice is that only one inference rule in the input file is in use, a rule for drawing conclusions, namely, ur-resolution. That choice is natural, at least to me, because you wish to encourage the reasoning program to deduce unit clauses, conclusions that say that some particular person holds, or does not hold, some specific job. But you might indeed wonder why no other inference rule is needed, for example, binary resolution or paramodulation, each of which focuses on exactly two clauses at a time. Well, I cannot give a good reason; I can only say that ur-resolution sufficed.

You next may notice that weighting is used, with assignments that (in effect) give preference to information deduced about Roberta, then Thelma, then Steve, and finally Pete. That preference causes the program to emphasize lines of reasoning, for example, that focus on an item about Roberta over a item that focuses on Pete. Such an emphasis is dictated by the puzzle itself, by the knowledge it gives you about the four people in the cited order. (The tricky part is not far away.)

Perhaps next you are curious about the two lists, usable and sos, and the placement of items on each. The elements of (the input) list(usable) are used only to complete the application of an inference rule, and not to initiate the application of an inference rule. In general, to aid your program from getting lost, you prevent it from spending its time on certain types of information that you suspect will play a far less effective role in solving the puzzle in focus or proving the theorem under consideration. Stated in a more positive manner, the eight items you find on list(sos) each reflect specific information about the four people, and the choice of the eight is motivated by the belief (guess) that the program should emphasize the role of each of the eight in the beginning. Further, to ensure that all of the items on the list(sos) are considered for lines of reasoning, you include the command set(input_sos_first). Although many items on list(usable) are not general, in the sense that they contain no variables, they are not thought to be key to driving the program's search for the answers to the jobs puzzle.

A closer inspection of the input file reveals that the idea is to emulate the table approach to finding out who holds which jobs. For example, the function *l* is to be interpreted as "list". In other words, you are making lists of the possible jobs, pj, for each person and the possible, PPL, people for each job on a list. So, in effect, you are making the rectangle of columns for people and rows for jobs. You are therefore now reading about the tricky part, the items that enable the program, when its reasoning begins to succeed, to place yes or no (in effect) in various squares, to "cross on or off", for example, possibilities. And, indeed subtle, you see three types of "equality", which are necessary: the usual equals, the equals for people (EQUALP), and the equals (EQUALJ) for jobs. As expected, you have reflexivity for each, such as EQUAL(x,x) for ordinary equals.

If you use the input file with OTTER, as it attempts to solve the jobs puzzle, you will find in the output file clauses like the following, clauses that show substantial progress is occurring.

261 [back_demod,240,demod,257,251,66,66] POSSJOBS(l(pj(Thelma,chef),l(pj(Thelma,boxer),end))).
301 [back_demod,286,demod,299,297,66,66] POSSJOBS(l(pj(Steve,nurse),l(pj(Steve,police),end))).

If you compare these two clauses with the correct answers (given earlier) to who holds which jobs, you will see how the input file, when used, eventually solves the puzzle. If, instead of puzzles, you enjoy areas of mathematics and areas of logic, I promise you that substantial success does occur—and has occurred since 1988 when McCune designed and implemented OTTER in approximately four months.

7. A Small Start, with the Promise of Great Excitement

You have been introduced to a bit of the elements of automated reasoning. Of those, to me, strategy is the most intriguing. As in poker and in chess and in activities of so many types, strategy is the key to winning. A charming feature of an automated reasoning program is its ability to proceed for days and weeks tirelessly, deducing millions and millions of new conclusions of which it may decide to retain as many as 7 million. A program such as OTTER makes no mistakes in the sense that each conclusion it draws follows inevitably and logically from the information supplied in the beginning. Of course, if you

happen to include a bit of erroneous information, then the program, trusting you, may deduce strange notions. If you omit a key item, for example, something that is obvious to you but not to a program that knows nothing outside of equality, then all may be lost. Indeed, if you tell OTTER to find many proofs by assigning a value such as 20 to max_proofs, I suspect you will be startled at the number of different proofs that might result—some of which might be correctly classed as bizarre.

As those of you who have browsed in various of my notebooks know, I have spent much time and energy in the pursuit of "shorter" proofs. I find it most satisfying when I find a proof shorter than that which the literature offer—and especially a proof that is shorter (and, to me, more elegant) than the proof supplied by a master. For example, when I found a proof of length 38 for Meredith's single axiom P(i(i(i(i(x,y),i(n(z),n(u))),z),v),i(i(v,x),i(u,x)))) for two-valued sentential (or classical propositional) calculus, I was exhilarated to find that my proof was four steps shorter than his proof of length 41 (applications of condensed detachment).

The excitement that I have experienced with an automated reasoning program awaits you. This notebook provides a small start. If you wish to go further, my other notebooks may be just what you need.