A Surprising Story of Research Discoveries: How Significant Advances for Automated Reasoning Occurred

Larry Wos

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

1. Prelude to Discovery

To those who have read one or more of my notebooks, I note that Ross Overbeek, I believe in 2007, was the wellspring for the set. Indeed, he insisted that much of my research, unpublished as yet, might be lost to the curious and to various researchers because nobody would ever deeply browse in my huge number of files.

On the one hand, my reason for adding this particular notebook to my previously written notebooks concerns the telling of possibly interesting stories. On the other hand, and perhaps more important, my motivation rests with the thought that perhaps some of you who read this notebook will find a rather different path—from that sometimes found in books—to research and to the discovery of mechanisms that, when implemented, have added to the power of one or more automated reasoning programs. For me, the program I rely on is William McCune's OTTER, a program that permits you to seek proofs of a number of theorems in a single run, even when the theorems have little or nothing in common. This narrative will consist of anecdotes that are drawn from my history, stories of how new ideas were spawned. As all would guess, I know my research history far better than I know that of any other researcher. Therefore, in an obvious way, this notebook is somewhat biographical.

As the years passed, I learned from the successes and disappointments of others. Perhaps some of you also sometimes learn in this fashion.

I began the study of a field then known as mechanical theorem proving in 1963, at the suggestion of William F. Miller, head of the (at-the-time) Applied Mathematics Division at Argonne National Laboratory. My study from then until now, here in 2015, I am sure was influenced by the time I spent at the poker table and at the pursuit of possible winnings from betting football games. In other words, I took chances in my research, chances based often on little evidence. Were I to set the type of example recommended by teachers and scholars, I would now strongly suggest that you study hard, examine the results of your experiments thoroughly, and contemplate for many, many hours. After reading the anecdotes I shall offer, you will justly conclude that such an approach was not mine, although it has been used successfully many times. Indeed, so often a single problem, a single experiment, led me to (a perhaps hasty) formulation of a new means to increase the power of automated reasoning programs. Am I encouraging those new to automated reasoning and those with much experience to so-to-speak jump to conclusions, which, of course, is obviously risky, but, as will be discussed here, occasionally rewarding?

If this narrative, when read, is to spawn new research into the formulation of new strategies to restrict or to direct a program's reasoning, almost demanded—at least for some of you—is a glimpse of a difficult problem to solve. Therefore, before the so-called biography commences, I offer the following deep problem for consideration.

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

The inference rule paramodulation—whose use enables an automated reasoning program to treat equality as if it is understood and, further, generalizes the usual notion of equality substitution—is indeed dangerous. (I shall introduce paramodulation in detail later with examples that will show how fecund its use can be.) In particular, from two items, perhaps two equalities, the program can often deduce many, many new items. Indeed, the use of paramodulation, though frequently key to success, can drown a program in new conclusions. What is needed—and here is the deep problem to solve—are strategies to effectively control its use. Two well-known restriction strategies do exist: one strategy has the program never paramodulate *from* a variable, and the other has the program never paramodulate *into* a variable. Even with the use of the two cited strategies, however, the program can sometimes spend far too much time on one pair of items, deducing too many useless conclusions. In most studies, the use of each of the given strategies is almost a requirement. After all, if you permit either type of paramodulation, from or into a variable, the corresponding application of paramodulation always succeeds—usually producing a conclusion that is not needed. Typically, you would prefer that your program did not drown in unneeded information. When I turn in Section 5 to a detailed discussion of paramodulation, I shall focus on another restriction strategy for paramodulation and, at that point, suggest an idea for possible future research, an idea that I may not have thought about until the writing of this notebook. But, with this sketchy prelude in hand, anecdotes are called for.

2. First Discovery: Unit Preference

In the early 1960s Alan Robinson wrote a paper introducing binary resolution and suggested an approach for using it to find proofs. His suggestion, if memory serves, amounted to level saturation, namely, have the program consider the input statements two at a time for all pairs, in order to obtain children, and then obtain all grandchildren, great-grandchildren, and so on. The input items are, technically, at level 0; their immediate offspring are at level 1; the grandchildren are at level 2. The basic idea is to explore all of level 1, then level 2, then level 3, and continue in this manner until at some level a proof by contradiction is found.

No preference is given to how complicated an item is, whether input or deduced. (In contrast, years later, Overbeek introduced weighting, which takes into account the actual or assigned complexity of an item or clause.)

In particular, if an item offers two choices, three choices, four choices, or more—to be exemplified—no special treatment is to be given based of the number of choices. For example, when represented in a language frequently used (the clause language), the equivalent of Kim is female or Kim is male is given no preferential treatment (with level saturation) over, say, the equivalent of Kim lives in Chicago or Kim lives in Detroit or Kim lives in Los Angeles or Kim lives in New York. The first of these two items, when expressed as a clause, yields a two-clause (for two choices), and the second yields a four-clause (four choices). If exactly one choice is found in an item, input or deduced, the corresponding clause is called a unit clause, for example, Kim is married. (Shortly after I entered the field, I posed an approach that emphasizes the role of unit clauses, discussed soon.)

At this point a respite is in order, one in the form of a question that might prove amusing. If you are expert in the use of binary resolution, you might prefer to quickly pass over the next few sentences.

When Robinson first offered binary resolution, it was defined to focus on two clauses, each of which contained an appropriate literal. The appropriate literals had to be opposite in sign and, in effect, contradict each other; in other words, when unification was applied, it must succeed. The resulting resolvent was obtained by removing the so-called appropriate, unifiable literals and then taking the logical **or** of all the remaining literals, if any. When Robinson first formulated binary resolution, he thought it to be refutation complete, meaning that, given a set of clauses that should lead to a proof (eventually), binary resolution would suffice to find a proof. Well, if memory has been maintained, Bill Davidon at Argonne National Laboratory gave Robinson an example of a set of clauses that, if treated properly, would lead to a proof—but not with the given formulation of binary resolution. Can you give such an example? (Later I shall give Davidon's example.)

Now, let us see what might happen if the number of choices is ignored and binary resolution is in use. For those new to the field, I note that the symbol " | " denotes logical **or** and the symbol "-" denotes logical **not**. Symbolically, if binary resolution is applied to the pair of clauses -a | -b | c and b | d, you get the clause -a | c or d. In other words, from two items each of which is a nonunit clause (contains more that one literal, subitem), binary resolution yields a conclusion that is more complicated than either premiss.

At the time, I felt that an approach based on level saturation would often drown a program before the goal was reached. I believed that the program would do well to emphasize the role of simple (unit) clauses for drawing conclusions—clauses that offer but one (so-to-speak) choice—rather than focusing on an item that just happened to be next in order. Further, I felt that the program would do well to attempt to deduce unit clauses if possible. With this approach the program would, if possible, apply binary resolution to a pair of items one of which is a unit and the other a two-clause.

Put symbolically, if the automated reasoning program had among its items a clause consisting of just the single literal c and one of the form -c | e, then it could immediately deduce the clause e with an application of binary resolution; that is, the program could draw this new simple conclusion. By deducing one unit clause after another, by drawing such new conclusions, the likelihood of eventually—but often sooner than with level saturation—deducing two unit clauses that are opposite in sign and that "contradict" each other will ordinarily be sharply increased. For a simple illustration taken from puzzle solving, with the given approach—called the unit preference strategy—the program might readily deduce CHICAGO(Kim) and soon deduce -CHICAGO(Kim), Kim lives in Chicago, and Kim does not live in Chicago. In other words, a proof by contradiction has been completed.

The unit preference strategy represents my first contribution to what would become (perhaps in 1979) automated reasoning. I did not arrive at the formulation of this strategy after careful and thorough analysis; it just seemed clear, intuitively, that the presence of unit clauses add to the chance of success. If others have had this so-to-speak spontaneous experience, I am unaware of such. Indeed, years later, Overbeek introduced the inference rule *UR-resolution*—a rule that seeks the deduction of a unit clause from a set of premisses all but one of which is a unit clause, with the remaining one a unit clause with one more literal than the number of unit premisses. The formulation of UR-resolution clearly merits praise. And I suspect it was not formulated in a matter of minutes or hours, though I never asked him.

3. One Problem, One New Strategy

I now come to the type of anecdote that was promised in Section 1, an anecdote whose significance is far exceeded by what resulted from the experience it captures. The time was early 1964. The program in use was brilliantly designed by Dan Carson; we worked together at Argonne National Laboratory. (It did employ the unit preference strategy.) The problem that spawned the new idea was taken from group theory, asking for a proof of the theorem that asserts commutativity to be provable for groups in which, for every element x, the square of x is the identity e. Some of you might correctly classify this theorem as a classroom exercise, which it was titled in a paper published in the middle 1950s. (When I mentioned this paper to my advisor, Reinhold Baer, he nicely suggested that I never publish a paper featuring such a small result.) Despite the lack of depth in the cited theorem, our attempt to prove it failed, failed because Carson's program ran out of memory; it could hold only 2,000 clauses—not a fault of his design, but a reflection of the size of computers in the early 1960s.

As I borrow from a story I told in Chapter 2 of the second book I wrote on automated reasoning, I note that Carson called me from the machine room and said, quite harshly, that we were done, finished. I asked for 45 minutes, with the intention of finding something that might overcome the given obstacle of deducing too many conclusions before a proof was found. He gave me 32 minutes.

My conjecture was that the program was being drowned because it was focusing too much on the axioms of group theory, and not enough on the items that are directly relevant to the theorem. (Intuitively, the axiom states that product is associative, that there exists a two-sided identity element—that is, when an element multiplied by it, nothing happens—and, with respect to the identity, that a two-sided inverse exists.) In other words, no special focus or preference was being given to either the statement $xx = e$ or the denial of commutativity, $ab \mathrel{!}= ba$.

Thus, a new strategy was formulated, called the set of support strategy. That strategy has the user of an automated reasoning program choose from among the input clauses one or more that is placed on a list, called the list(sos) in William McCune's program OTTER (as well as many of its predecessors). With the set of support strategy, a program is prevented from drawing a conclusion all of whose premisses are in the complement of the initial set of support. In the theorem whose consideration gave rise to the strategy, the program was not permitted to draw a conclusion from a set of axioms (of group theory), for none of the axioms were placed on the (initial) set of support. The curious reader immediately asks two questions. First, what happened? Second, what action is taken with newly deduced conclusions that the program decides to retain?

The computer in use in 1964 at Argonne was an IBM 704, indeed not very powerful by today's standard in 2015. Nevertheless, only 3 CPU-seconds sufficed to yield the desired proof. (By the way, when I told Carson about the new idea, after those 32 minutes, for the first time in my dealings with him, he showed substantial excitement.) Yes, we were pleased and intrigued by the possibilities of relying on this new strategy. As for the second question, the strategy has the program place newly deduced-and-retained clauses in the set of support, items to be used to initiate lines of reasoning when chosen from that list.

4. A Single Problem Leading to Much Study by Others

Perhaps some of you have spent much time in the study and use of *complete sets of reductions*, sets of equalities possessing certain termination properties. For the historian—and for those who may not have known it but find it more than piquant—the idea of complete sets of reductions was formulated about three years later than was the formulation of *demodulation*, the focus in this section. What is demodulation, and why is it of use? Demodulation is the process of canonicalizing and simplifying expressions, clauses for example, with equalities; such equalities can be in the input, and they can be found during an attempt to reach a goal. By way of illustration taken from mathematics, if the input contains $0 + x = x$ and if the program deduces and temporarily retains $b + (0 + a) = c$, then, with the given demodulator, the program will retain $b + a = c$. Or, in everyday language, if the program has the demodulator mother(Mother(x)) = grandmother(x), and if the program deduces and temporarily retains mother(Mother(Gail)), then, with the given demodulator, the program will retain grandmother(Gail). Of course, for the sophisticated researcher in automated reasoning, the cited retentions in the two illustrations are subject to other possible constraints.

But in focus is the relation of demodulation to a single problem. Specifically, in the late 1960s, the Argonne group was making a bit of progress seeking proofs in various areas of mathematics. For example, we studied some easy-to-prove (for a person) theorems from ring theory, but often we were forced to adjoin possibly needed lemmas. When we tried the theorem from number theory that asserts that the square root of 2 is irrational, our program was unable to complete a proof. To find a proof with the program, we were forced to include a powerful lemma. My cursory examination of the failure, an examination of 2,000 clauses, revealed items of the sort $0 * a * 0 = a$.

I quickly conjectured that such trivial items dominated the retained conclusions, that, if read carefully, many, many such items would be found. To check the conjecture, I read more carefully the 2,000 clauses and found that I was in error; indeed, only 61 such trivial clauses were present among the 2,000. I of course accepted that my conjecture was (so-to-speak) false; but rather soon I conjectured that the removal of such clauses, if viewed as roots of trees, would lead to a set of clauses with far fewer than 2,000. And demodulation was born.

Indeed, with the inclusion of equalities such as $0 * x = x$ and $x + 0 = x$, the next experiment did show that almost all of the 2,000 clauses were not retained. More fully, as each new conclusion was drawn, it was demodulated with such equalities to canonicalize and simplify it before considering it for retention. Demodulation can yield astounding bits of information such as $a = a$, subsumed by $x = x$. Without the use of demodulation, my experiments in areas of mathematics and logic would never have yielded much success, areas such as group theory, lattice theory, and classical propositional calculus. As for complete sets of reductions, I would never have followed a path that led to that charming area of study; such was and is not my type of thinking.

Demodulation was and is used for activities other than canonicalization and simplification. In particular, in the context of strategy, I often use demodulation as a purging strategy, to sharply reduce the number and (sometimes) type of retained conclusion. For a perhaps unexpected example, in various areas of logic, a desired proof can often be found much sooner by purging deductions the contain an expression of the form $n(n(t))$ for some term t. To connect your thinking to items you may have read in the past, the idea is to avoid double-negation terms. You purge unwanted conclusions often by demodulating such to "junk".

5. Generalizing Equality Substitution

I turn now to the topic of paramodulation. I note that a single problem or theorem played no role in its formulation, from what I can recall. Below, using material from my Primer notebook, I present three examples as a quick review of paramodulation. 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.

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 preceding 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 preceding three clauses, *from* the first *into* the second, to logically deduce the third (again without producing any intermediate clauses).

Where did paramodulation come from, if not from the consideration of a single problem? The principal impetus was a paper by Robinson, in which he suggested that two problems merited study, one focusing

on equality and one on set theory. Until then, equality was treated as just another relation, not built in at the inference level. Another impetus, almost surely, was the earlier introduction of demodulation. Indeed, the first example just given exemplifies demodulation as well as paramodulation. With demodulation, during the unification, variables can be replaced in the demodulator by terms; but such an action is not permitted for the expression being demodulated. In paramodulation, especially illustrated with the third example of that inference rule, (during unification) nontrivial replacements of variables can be made in both the *from* and the *into* items. Thus, you see, particularly with the third example, how paramodulation generalizes the usual notion of equality substitution.

Since the program can, if not restricted, paramodulate into any expression or subexpression of the *into* item, even if variables are excluded, you can imagine how fecund the use of this inference rule can be. In Section 1, you read about two restriction strategies, each focusing on a variable, preventing the program from paramodulating *from* or *into* a variable, respectively.

I now offer you another restriction strategy, as promised in Section 1, that is provided in McCune's automated reasoning program OTTER. A glance at the three given examples may aid you in a fuller understanding of the strategy.

With OTTER, you can restrict the actions of paramodulation by blocking paramdulation from the right or the left side of an equality; in parallel (so to speak), you can take similar actions with regard to *into*. The following suffice.

```
clear(para_from_right).
clear(para_from_left).
clear(para_into_right).
clear(para_into_left).
```

I have used this type of strategy in various studies, including combinatorial logic, and in various combinations. What I do not recall seeing is the following, an idea that might be a seed for possible future research (as alluded to in Section 1). The idea involves giving a reasoning program a set of templates or weights to govern the type of *into* term that is permitted for paramodulation, or (similarly) with regard to the *from* argument. Equally, you take similar actions for avoiding certain classes of *into* term. For a simple example (which might not prove to be effective), you might have your program never paramodulate into a term that contains nested functions. For a second example, you might block the use of paramodulation when the *into* clause contains more than twenty symbols. For a related example, you might take corresponding actions with regard to the *from* argument. As noted, I do not recall any member of the Argonne extended group experimenting with this notion. So you now have a deep problem to consider, namely, finding a way to sharply control the use of paramodulation. Overbeek's weighing strategy might mesh well with this proposed strategy.

6. A Direction Strategy

In ring theory, you can prove commutativity for rings in which the cube of every element x is x, $xxx = x$. Not surprising, a number of quite different proofs exist for this theorem. Particularly of interest in abstract algebra, you can replace the cube by n for $n >= 3$. When I brought the theorem featuring the cube of x to Steve Winker, he—as I expected—supplied a proof quickly. If you examine various proofs of this theorem, for the cube in particular, you will find that the special hypothesis, $xxx = x$, plays a key and frequent role. In other words, many of the steps found in, for example, Winker's proof also use $xxx = x$ as a hypothesis.

When I made this not-profound observation, I almost immediately thought that an automated reasoning program would benefit from some strategy that caused it to visit, revisit, and re-revisit the special hypothesis of any theorem it was trying to prove. And the hot list strategy was born. Again, a single problem, a single theorem, led to what turned out to be an advance for the field of automated reasoning.

This strategy meshes well with an approach based on Knuth-Bendix. Thanks to McCune, its general power is exemplified by experiments I have completed in multivalued sentential calculus, classical propositional calculus, Tarskian geometry, and other areas. My original formulation of the hot list strategy was solely in the context of equality-oriented reasoning. McCune implemented it for OTTER, extending the hot

list strategy to all inference rules, even those in which equality plays no role; in addition, he formulated a dynamic version of the strategy. Specifically, the program has the power, if certain parameters are set, to adjoin during a run new elements to the initial input list. McCune may have been the one, rather than I, who generalized the strategy to enable the program to iterate (when the appropriate value is assigned to heat), to revisit, and to re-revisit the use of elements in the hot list when a new conclusion was retained. Were I to seek a proof of the cited ring theory theorem with the use of OTTER, I would place $xxx = x$ in the initial hot list. At the moment, I do not have a recommendation for the type of conclusion that, when retained, should be adjoined to the hot list.

7. A New Class of Inference Rules

In the early 1980s, I wrote a book with colleagues in which a puzzle was featured, called "the jobs puzzle". The puzzle tells about four people and the eight jobs that they possess, and it provides clues to enable you, or a program, to determine which of the four people held which job. One of the clues asserts that the job of nurse is held by a male. Now if you were presented with this puzzle, you would immediately deduce that Roberta, one of the four people, was not the nurse; that is, you would draw this conclusion in one step.

The typical reasoning program would reach the same conclusion; but, unfortunately from the viewpoint of possible inefficiency, two steps would be required. Indeed, the program would first deduce that Roberta is not male, an item that it would then have available for drawing possibly many conclusions of little or no value to solving the puzzle. After deducing that Roberta is not male, that deduction would be used to deduce that Roberta does not hold the job of nurse. Perhaps such so-called intermediate conclusions are merely a nuisance and not worth bothering about. On the other hand, both UR-resolution and hyperresolution have the advantage over binary resolution by taking larger steps, steps often of greater significance, avoiding so-called intermediate deductions that would be made with a replacement of either of the rules by binary resolution.

The triviality of the example from the jobs puzzle could easily suggest that the implied generalization of UR-resolution is merely a curiosity. But that single example caused me to formulate linked inference rules—linked UR-resolution, linked hyperresolution, and the like. While standard UR-resolution requires that literals of the nucleus be removed directly with unit *satellites*, linked UR-resolution permits literals in the nucleus to be removed either with unit satellites or with nonunit *links*. One could intuitively say that the literal MALE(nurse) is removed by the nonunit clause -FEMALE(x) | -MALE(x), by the link -FEMALE(x) | -MALE(x), which links to the unit clause FEMALE(Roberta). For a more persuasive example, consider the case in which six constants are present, a through f, and the program is told that $a < b$, $b < c$, ..., and $e < f$. With a linked inference rule, the program can deduce in one step that $a < f$ and avoid deducing all the obvious intermediate inequalities. Summarizing, just as hyperresolution and UR-resolution enable the program to take, sometimes, much larger steps than with binary resolution, linked inference rules go even further in that direction. So, again, one problem led to a new idea, the cited linked inference rules, modified and extended by Robert Veroff.

8. Another Restriction Strategy

Immediately before Section 5, I cited a strategy that avoids so-called double-negation terms. The idea is this: When you are studying some area of logic in which negation is present, denoted by the function n, typically in the literature on that area you find formulas containing expressions such as $n(n(t))$ for some term t. You might in your studies, as I did, conjecture, or know, that many, many conclusions are drawn from this double-negation class. Further, you might suspect that their presence is interfering with the program completing the assigned task, say, of finding a desired proof.

When McCune and I were studying an area of logic known as infinite-valued sentential calculus, MV, I wondered about finding proofs in which no double-negation term was present. In other words, I was interested in the value of a restriction strategy that restricts the program from retaining conclusions in which a type of term was present. Such a strategy might indeed prove useful in your studies. You might be able to identify some class of terms to be avoided such that the avoidance promotes a sharp increase in efficiency.

When I had OTTER avoid double negation, I often found proofs in far less CPU time. Even better, sometimes I was presented a proof that had, without the strategy, eluded me in *MV* and in other areas of logic. (By the way, Michael Beeson and Robert Veroff formulated and proved interesting theorems that focus on conditions of a logic that guarantee the existence of double-negation-free proofs.)

At this point, some of you may recall the use of demodulation to block one or more chosen formulas, an action that is somewhat similar to the avoidance of double negation. An application of such an activity, the blocking of one or more equations or formulas, concerns the case in which you have a first proof and wish to find a second proof, one that avoids a lemma, deemed undesirable, present in the first proof. In other words, you are employing an instance of another restriction strategy, restricting the retaining of some unwanted equation or formula, reminiscent, perhaps, of restricting the program from relying on some type of term such as the avoidance of double-negation terms. This strategy has not been named, and it is actually a use of demodulation quite different from the original motivation of simplification and canonicalization.

9. A Domain-Oriented Strategy

I include the following strategy especially for those researchers who are focusing heavily on some particular area of mathematics or logic. The area in focus here is combinatorial logic, an area that McCune and I studied intensely. That logic, briefly, concerns various combinators such as *B* and *W* whose actions are specified as shown below and which are assumed to be left-associated unless otherwise indicated.

Bxyz = x(yz), for all x, y, and z
Wxy = xyy.

A fixed point combinator, by definition, is a combinator F such that $Fx = x(Fx)$ for all x. One of the questions McCune and I studied asked about the possible existence of a fixed point combinator expressed purely in terms of *B* and *W*. Such do exist, first proved by Richard Statman in February 1986. McCune and I found many fixed point combinators in terms of *B* and *W*. More generally, based on a key observation of McCune's, I formulated what I called the kernel strategy, whose use enables a program such as OTTER to quickly find fixed point combinators, if they exist, given the combinators to be used. Exceptions to the method do exist, but few indeed.

You thus have an example of a domain-specific strategy, a strategy whose use led directly and indirectly to our answering—some with OTTER—a number of open questions. For clarity, I note that the kernel strategy is unlike the restriction strategy focusing on avoidance of double negation in that the former is oriented toward combinatory logic exclusively, whereas the latter is useful for many areas of logic. Indeed, one can formulate an analogue to the double-negation restriction strategy for areas in which equality is dominant. Specifically, where appropriate, you might have the program block the retention of conclusions that contain some expression of the form $inv(inv(t))$ for some term t.

10. A Direction Strategy

I have a deep affection for restriction strategies. To complement strategies that restrict a program's reasoning, however, we also need strategies that direct its reasoning. Overbeek formulated weighting to guide a program in various ways, one of which is to enable it to focus on profitable items. In particular, with weighting you can cause your program to choose, from among its retained items, an item to initiate a new line of reasoning, where that item might ordinarily be forced to wait a long, long time before coming into focus. For an example, with appropriate weight templates, you can have your program choose next an item that relies on twelve symbols even though many items relying on ten symbols are yet to be chosen. In other words, you are employing a direction strategy, directing the program to focus on an item, in preference to many others, because your knowledge and intuition suggest that the chosen item will lead to one or more better conclusions.

A few years ago, the logician Dolph Ulrich brought to my attention a single axiom due to Adrian Rezus. Indeed, when Ulrich constructed for me the following Rezus-style 93-symbol formula that is itself a single axiom for the implicational fragment of *R*, I began wondering about how one could prove this formula to be a theorem of that logic.

```
P(i(i(i(x,i(i(i(y,y),i(i(z,z),i(i(u,u),i(i(v,v),i(x,w))))),w)),i(i(i(i(i(v6,v7),i(i(v7,v8),i(v6,v8))),
  i(i(i(i(i(v9,i(v9,v10)),i(v9,v10)),i(i(i(v11,v12),i(i(v12,v13),i(v11,v13))),v14)),v14),v15)),v15),
  i(i(v16,i(i(i(v17,v17),i(i(v18,v18),i(i(v19,v19),i(i(v20,v20),i(v16,v21))))),v21)),v22))),v22)).
% Rezus-style single axiom for RI
```

The goal is to have OTTER rely solely on condensed detachment to eventually find a proof that the given Rezus-style formula is a single axiom for *RI*, the implicational fragment of *R*. The axioms I focused on for *RI* are the following.

```
P(i(i(u,v),i(i(w,u),i(w,v)))). % B
P(i(i(u,i(v,w)),i(v,i(u,w)))). % C
P(i(u,u)). % I
P(i(i(u,i(u,v)),i(u,v))). % W
```

The objective is to prove that the Rezus formula is in fact a theorem of *RI*—is derivable from *B*, *C*, *I*, and *W*. Because the Rezus formula has length 94 (in symbol count with the predicate symbol), this task is ordinarily most formidable, in contrast to using the Rezus formula to derive the four given formulas.

Perhaps a bit more will illustrate the difficulty by trying to use those four small (in length) axioms to derive the Rezus 93-symbol formula. In many automated reasoning programs, two mechanisms exist for choosing where next to focus in order to obtain a retained element to be used to *initiate* a line of reasoning. One mechanism is level saturation, an approach suggested by Robinson in his original paper (even before publication) on binary resolution. With level saturation, the program finds all level-1 items, then level-2, then level-3, and proceeds until a proof is found, where the input items are at level 0 and those at level n+1 are obtained from a set at least one of which is at level n. When I began my study of the Rezus formula, I naturally had no idea of how many levels would have to be examined. I have in my files proofs of level 70, so a level-saturation approach seemed absurd.

The second mechanism, complexity preference, is based on the complexity of retained items; usually the least complex is chosen for initiating a new line of reasoning. But—and you have already seen the light—a proof of a 93-symbol formula would almost certainly involve formulas of complexity 50, 60, 70, and more. So, a complexity-preference approach was ruled out by me. Since I could, of course, combine the two approaches with McCune's ratio strategy, I did have a third choice; but this, too, was quickly deemed untenable.

What remained—especially in view of my continuing to give examples of a single problem leading to a breakthrough—was to formulate a new strategy. Indeed, the *subformula strategy* was born, whose birth resulted from the properties of the Rezus formula. Briefly, the basic idea is to take all the nontrivial subexpressions of the complex element under study and, in the input file, include for each a weight template with a small assigned value. Weight templates are used to direct the program's reasoning toward more profitable paths of inquiry, if all goes well. The trivial expressions are constants and variables. And yes, after a number of experiments, on February 8, 2008, OTTER found a proof, one that deduces from the four given formulas the Rezus formula. The proof took a bit more than 19,000 CPU-seconds, but, perhaps surprisingly, it was not a long proof; indeed, the proof has length 18 deduced steps of condensed detachment.

11. Another Direction Strategy

For the next example of the formulation of a new strategy, I turn to a memory of a workshop held at Argonne National Laboratory. Apparently prompted by results presented at the workshop, one of the participants, the logician Dana Scott, subsequently issued a challenge for us. The challenge was to prove 68 theorems from two-valued sentential (or classical propositional) calculus, theorems that Lukasiewicz calls theses. Lukasiewicz offered the world the following axioms for this area of logic, where the function *i* denotes implication and the function *n* denotes negation.

```
%  Luka 1 2 3.
P(i(i(x,y),i(i(y,z),i(x,z)))).
P(i(i(n(x),x),x)).
P(i(x,i(n(x),y))).
```

McCune and I were eager to meet Scott's challenge, and we began our attempt. As I recall, OTTER quickly found proofs of 33 of the 68. Then, at McCune's suggestion, a parallel-processing version of OTTER was used, resulting in proofs of 47 of the 68. You might say, not bad; but we guessed that Scott would be disappointed. What we did next, therefore, was turn to the *resonance strategy*, which you may have read about in my other notebooks. Specifically, we placed each of the 68 theses, with a small weight, in a list that directs the program in its reasoning. To be totally clear, such templates do not get the value **true** or **false**, nor are such templates treated as part of the problem description. Here, for example, the 68 resonators are used to direct the program in its choice of where next to focus its reasoning, which item to choose next for initiating a line of reasoning. In this specific case, the idea is that when and if one of the Lukasiewicz theses was proved, very soon that formula would be chosen next to reason from.

Some of you, understandably being suspicious, might suggest that the program was merely (in effect) proof checking the research of Lukasiewicz. More pointedly, you might feel that most likely the 68 theses formed the outline of a larger proof and, therefore, this use of the resonance strategy was merely following the outline. As a matter of fact, after a few months, one of my colleagues posited this precise view.

The obvious path to put aside such views or fears is to try the new resonance strategy in other areas of logic. Further, a good test would be to choose a different field of logic and use, for resonators, formulas that are conjectured to not form an outline of a proof. For example, you could experiment with another area and use—perhaps surprise—the same 68 templates that correspond to the Lukasiewicz theses. I, for example, used the resonance strategy in studies of Lukasiewicz's infinite-valued sentential calculus, which relies on implication and negation. Of course, the area, in this context, to be chosen had best rely on implication and negation if the Lukasiewicz theses are to be used as resonators. More generally—and I did exactly what I now suggest—you could try templates from one area in another area and see what results.

And indeed, my experiments have repeatedly indicated that the resonance strategy is powerful and that its use does not correspond merely to proof checking some given implied outline. My colleague did finally agree that the strategy was far from such an interpretation. By the way, I believe it safe to report that, upon receipt of all 68 proofs, Scott was pleased—and, yes, quite impressed.

The resonance strategy also proved useful in my studies of what I call elegant proofs. Informally, three aspects of elegance can be quickly cited. Proof length is perhaps the most obvious aspect. Everything being equal, the shorter the proof, the more elegant. The second property concerns the structure of a proof, specifically, the nature of the terms present in the deduced steps. For example, one might seek a proof in which absent are terms of the form $n(n(t))$ for any term t, where the function n denotes *negation*. The third property I cite might most interest certain researchers. This property is *compactness* of the proof. Rather than a formal definition, the following illustration suffices, for you can extract from it (if desired) the appropriate formalism. Let the theorem **T** under consideration be of the form P implies Q and R and S. For example, let P be the Lukasiewicz axiom system consisting of $L1$, $L2$, and $L3$, and, respectively, let Q, R, and S be theses 18, 39, and 49 (the Church axiom system). If by hand or with a program you find a proof of **T** such that the proof is a proof of exactly one of Q or R or S, then you have a proof that is *compact*. Of course, to be a proof of, say, R requires that the last step be R and that all steps be needed in the proof. Therefore, a compact proof of **T** that completes with the deduction of R must contain as subproofs proofs of each of Q and S.

And, in answer to a possibly pressing question, the resonance strategy has proved useful for various studies in which equality plays the dominant role, such as studies of areas of abstract algebra. I have no idea about the value of the resonance strategy for solving puzzles with an automated reasoning program, when the puzzle is a hard puzzle.

12. The Promised Answer

Have you produced the example for binary resolution that shows the need for an additional inference rule, when binary resolution is defined to focus on a single literal in each of two clauses? Well, thanks to Davidon, who read of Robinson's original formulation of binary resolution, most (perhaps all) automated reasoning programs offer an inference rule known as factoring. The following set of clauses is unsatisfiable but cannot be proved to be so by using binary resolution as it is usually defined and typically programmed.

P(x) | P(y).
-P(u) | -P(v).

With factoring, from the first of the two given clauses, the program deduces P(x), and from the second it deduces -P(u), and all is well with the just-found contradiction.

13. Single Problem, Multiple Successes

I have titled this final section Single Problem, Multiple Successes (SPMS) in a sort of play on SPMD (single program, multiple data)—the most popular style of parallel programming today. Indeed, I have offered you in this notebook various examples of how focusing on a single problem or theorem led to advances in automated reasoning. Arguably, the type of discovery featured here is far from the norm. Nevertheless, I hope to inspire you to emulate or modify the research approach I have relied on.

To be clear, in contrast to so many of my contributions, I do not deny that years of research may sometime be needed before the results lead to a new and powerful approach or methodology. Finding a long-sought proof is just that: sometimes years elapse before the desired goal is reached. For one example, a huge number of experiments were featured in my own research before finding a 38-step proof showing the Meredith 21-letter formula (in classical propositional calculus) to be a single axiom, in contrast to his 41-step proof. For a second example, decades elapsed between my first effort at determining the status of the formula XCB in equivalential calculus and the success in 2002, when Ulrich and I showed that formula to be a single axiom.

I now offer two examples that do not focus on a single theorem, although perhaps focus on one type of problem, each idea occurring based on thousands of experiments. First, the methodology for finding shorter proofs was born perhaps in 1992 but continued to mature over the next fifteen years. Rather than focusing on a single theorem, I studied many, many theorems, often taken from various areas of logic. One of the features that I formulated concerns the use of demodulation, in contrast to its use for simplification and canonicalization. Briefly, the central idea is to block steps of a given proof, one at a time, with the hope that a shorter proof than that in hand would be found. Quite charming was the discovery that, occasionally, a shorter proof is found all of whose steps are among the so-called starting proof, that which was known before demodulation was used in the cited nonstandard manner to find a shorter proof. Naturally, you might wonder how this can be. The key clue rests with the following illustration. Imagine that you have, say, a 60-step proof, whose fortieth step is derived from steps 6 and 20. Further, imagine that step 20 is not allowed to be used. But, in this illustration, step 40 can be derived from steps 8 and 32. In other words you can avoid relying on step 20 and still find a proof.

Second, the cramming strategy may have had its wellspring in my interest in finding shorter and ever-shorter proofs and—perhaps more deeply—in my pursuit of answers to open questions. With cramming, in the simplest of cases, when the goal is a proof of a conjunction, you take a subproof of one of the members and *cram* or force as many of its proof steps into subproofs of the remaining members of the conjunction. Cramming typically, but not exclusively, plays a key role in finding a shorter proof and, occasionally, in answering an open question.

Certainly, as these various examples show, not all new contributions arise in a flash of insight. But, as I have indicated, very, very often in my own research such contributions do arise from focusing on a single example or theorem or problem. Further, in so much of my work, that focus has led to the desired conclusion within a short time, rather than in months and months. And perhaps you will find yet another example when I write my next notebook for this website.