papers/notebooks/program.enhancements
created 09-11-2009
revised 09-05-2010

Future Enhancements for Automated Reasoning Programs

Larry Wos

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

1. Visiting New Ideas and Research Topics

Would you enjoy evaluating some (probably) new ideas for enhancing the power and scope of an automated reasoning program, a program that features first-order logic? Do programming challenges appeal to you? Is the avoidance of a huge amount of iteration of interest? Would you like some notions that you can apply to hone your reasoning skills, unaided by a computer? Are you looking for a research topic? Do you find fascination in witnessing a program modify, without your intervention, its approach as it attempts to complete a given assignment? If your answer to any of the posed questions is in the affirmative, this notebook might provide you stimulation, entertainment, and, just perhaps, material that will lead to a paper to publish or a chapter to write.

In that regard, I pause for a bit to present a short note from Ross Overbeek.

A Short Note by Ross Overbeek

In this notebook Larry discusses a number of protocols for restricting the search made by an automated reasoning program such as OTTER. In the old days, we would have viewed such suggestions somewhat skeptically on the grounds that it took a great deal of effort to actually test such conjectures, and many of the conjectures made by senior people did turn out to be less than fruitful. I believe that the comments Larry is making here should be treated seriously for the following reasons:

1. Larry has grounded his comments in a huge number of runs that he has submitted and studied. Attempting to solve one problem after another does induce some level of insight, and Larry has been as successful as anyone else that I know of in making programs actually produce proofs.
2. It would take relatively little effort to test his ideas, given the advances in available programming languages. with a language such as Perl or Python, it would be relatively trivial to create an automated protocol (say, as a Perl script) in which
   a. input is prepared for OTTER from an abstract representation,
   b. an OTTER run is formulated in a way that causes it to run for some short period before it is interrupted,
   c. the output of the OTTER is parsed, and new parameters are constructed to reflect the outcome of preceding short runs, and
   d. steps b and c are iterated until a proof is acquired.

In Perl or Python, it is easy to generate OTTER input, and it is easy to parse the output.

This work was supported in part by the U.S. Department of Energy, under Contract DE-AC02-06CH11357.

I sincerely believe, were I a doctoral student entering the field, that I would create a test set of theorems that I considered quite difficult (but provable), I would formulate a set of strategies reflecting my understanding of what Larry was proposing in this notebook, and I would test them and tabulate the relative behavior of the differing protocols. My guess is that this would be a direct and simple path to a doctorate, but that is only a conjecture. I do speak from some experience, though. I did formulate a system to prove a challenge Larry posed almost 40 years ago; I was successful, and it did get me a doctorate.

Ross Overbeek

Resuming the Narrative

As you likely expect, the automated reasoning program I rely on is W. McCune's OTTER. However, what is discussed here is relevant to reasoning programs in general and, for that matter, to reasoning on your own.

Noting that some background will shortly be supplied, I ask you to consider one of the objectives of the material covered here. Specifically, in my iterative approach aimed at completing an assignment, some runs complete early during the night, leaving the program waiting for my next experiment. Would it not be more than pleasant—fine rhetoric?—if the program could simply proceed on its own, avoiding the waste of time that occurs while it waits for me, or for some other researcher? An exciting prospect to contemplate, as my esteemed colleague R. Overbeek would agree! The program I envision will, if so instructed, examine its results (partial in most cases), make choices about how to modify its approach to increase the likelihood of success, and then resume its attack. Although notions of this type have been addressed, what is needed are metrics that the program could use to decide when to change its approach and, then, where to go next. In no way am I suggesting that researchers will be, eventually, unneeded. That will not occur for centuries, if ever! However, so often the hours of the night are wasted with the program sitting idle. Instead, with certain enhancements, perhaps the program could, so to speak, learn from its successes and failures and proceed on its own, based on various analyses and criteria. This mythical, at this time, program could (in effect) conduct a series of experiments, each based on what had been occurring, just as I now do with a series of separate experiments.

With such a program, or with programs that do not offer self-analytical capability, even more strategy is needed than is available today, some to restrict reasoning, some to direct it, as well as other types of strategy. A few remarks are in order to provide background, especially for those less familiar with various types of strategy that (in my view) are crucial for attacking deep problems and hard questions. Whether your interests mesh with mine, the study of numerous aspects of mathematics and logic, or whether your interests are elsewhere, for example, verification or design, I strongly suspect that the role of strategy may not have been emphasized as it is in this notebook. Whatever your area of study, when an automated reasoning program plays a key role, its power (obviously) matters much. In my research, I typically rely on an iterative approach. Almost always, I give advice to the program I use (as noted, McCune's OTTER is the automated reasoning program I rely on). The advice may focus on R. Veroff's hints, on resonators, on the hot list, on the choice of which elements to place in the initial set of support, and more. Specifically, I expect that, to reach my goal, I will be required to make a series of experiments, later ones building on results obtained in earlier ones. And just as chess players, poker players, and (some) football players rely on strategy—for substantial success—I will use various types of strategy.

In automated reasoning, the majority of available strategies are restriction or direction. Why does a powerful program need both types of strategy, whether the axiom system of concern is small (as is often true in areas of mathematics or logic)or large (as is the case typically in database problems).?

The need for strategy rests with the (perhaps surprising to those who have not experienced such) huge number of paths that can be pursued and the gigantic number of conclusions that can be drawn in search of the desired proof or other goal. Most exciting to me are strategies that restrict the reasoning of a program. Clearly, if unrestricted, when the set of axioms is very large, the set of conclusions to be examined can grow wildly. What may not be so obvious is the exponential growth that can occur even when the (initial) set of axioms is small. Indeed, I have recently in studies of intuitionistic logic, with an axiom

system consisting of but ten formulas, obtained the sought-after proof only after the generation of almost 700,000,000 conclusions.

Of the strategies that I use to restrict a program's reasoning, the *set of support strategy* has proved (over decades) to be the most powerful. In that strategy, typically, you begin by partitioning the clauses that define the problem into three classes, the basic axioms, the additional so-called special hypothesis, and the denial of the conclusion. For example, if the goal is to prove commutativity in rings in which the cube of x (for every x) is x ($xxx = x$), the special hypothesis simply consists of one equation, namely, $xxx = x$. The denial of the conclusion similarly consists of one item, say, *ab != ba*. The basic axiom set consists of whatever axioms you prefer using that capture ring theory.

Since the idea behind the introduction of the set of support strategy was to prevent a program from exploring the entire theory from which the question or problem was taken, you typically place the basic axioms on one list (the usable list for McCune's automated reasoning program OTTER) and instruct the program to *never* apply the chosen inference rule(s) to a set of hypotheses on that list, in other words, among the basic axioms. Given that proviso, you can either place the elements of the other two classes together on the list of items that can be used to initiate an application of an inference rule (the set of support list for OTTER), or you can have that list consist solely of the special-hypothesis items. Instead, you could rely on so-called reasoning backward by placing the denial clauses in the initial set of support and placing the special hypotheses in list usable. Two additional cases in particular merit mention.

In one of the two extremes, which occurs in various areas of logic, you typically place all of the axioms on the set of support list and place the clause for the inference rule (often condensed detachment) on list(usable). At the other end of the spectrum, you have problems that occur in database inquiry. In such problems, the choice of which items to place on list(sos), the set of support list, may not be so clear. For an example quite unlike that focusing on ring theory, and one that is somewhat reminiscent of database inquiry, the following puzzle (from everyday language) serves nicely for illustrating an appropriate use of the set of support strategy.

> 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?

For this puzzle, which you might enjoy solving on your own, a typical use of the set of support strategy would place on list(usable) general information such as facts that include "husbands are male" and "everybody is male or female, but not both". Such an action prevents your program from reasoning about general properties of people, rather than focusing on the puzzle to be solved. (The given puzzle is sometimes known as the "jobs puzzle"; it is sometimes offered to gifted children. Nevertheless, you might find solving it requires a bit of thought.)

As for an example of the use of a direction strategy, in the context of the given jobs puzzle, (with OTTER) you can have the program focus heavily on any deduced conclusion when and if it involves one of the four people.

With strategies that include the two exemplified, I have (as discussed in various notebooks on my website, automated reasoning.net) been rewarded with the discovery of many, many new proofs of interest to other researchers. But, greedy as I am, I often experience impatience while waiting for a given experiment to terminate. Sometimes, I find that, in the middle of the night, a most useful conclusion was drawn, followed by a number of unprofitable hours of computation. I would like a program to be able to recognize

the value of the conclusion and take appropriate action immediately and, most important, on its own. Equally, if an experiment is evidencing little or no progress, I would like a program to realize that a change must be made, and immediately make one or more changes. In this notebook, I discuss various enhancements in the spirit just stated.

2. Tuning the Initial Set of Support List

The nature or content of the initial set of support can profoundly affect the likelihood of success. If too few items are present, then a vital path may fail to be explored. If too many items are present in the initial set of support, so many paths of reasoning may be explored that far too much CPU time is consumed, so much so that the goal is never reached. Each of these observations merits comment.

In the context of being too small, consider the case in which you are asked to prove commutativity for rings in which $xxx = x$. If the corresponding clause (for $xxx = x$) is placed in list(usable) for OTTER, for example, then key conclusions may fail to be drawn because they require the clause to initiate the needed application of the appropriate inference rule. Yes, its presence will enable the program to complete deductions; but if, just perhaps, a proof required the use of $xxx = x$ to be used to initiate some line of reasoning, its absence from the initial set of support could prevent the program from ever finding a proof of commutativity.

For an everyday language example, an example that might indeed lead you to produce a far more complex database example, the jobs puzzle will serve nicely. If the clause that corresponds to the bit of information asserting that "Roberta is not a boxer" is present, but not in the initial set of support, then a possibly crucial line of inquiry may not be explored. Indeed, the needed bit of reasoning that uses this fact might be required to begin with this fact. In other words, the puzzle may remain unsolved by the program because certain key items are not deduced.

In the opposite case, the initial set of support may simply be too large. For example, an inexperienced individual might place all of the information (that captures the problem to be solved) in the initial set of support. Indeed, before the set of support strategy was introduced in the early 1960s, a paper was written that advised the use of the following approach to proving theorems. The approach, one of level saturation or breadth first, had the program simply deduce all of the immediate children of all pairs of input statements; such are called level-1 deductions. Then all level-2 deductions (grandchildren) were to be made, from a set of level-1 statements with level-1 statements and from level-1 statements considered with input statements (those of level 0). Unfortunately, many proofs require information of level 10 or far greater. Now, if you had placed, say, thirty items at level 0 in the initial set of support, the program likely would drown in conclusions of level far smaller than level 10. In the context of rings in which $xxx = x$, if you placed all of the information in the initial set of support, which would include all of the axioms for a ring, the program might spend far too much time exploring ring theory as a whole, rather than focusing on the theorem to be proved. If, in the context of the jobs puzzle, you placed all of the thought-to-be relevant information in the initial set of support, the program might study far too long deductions about marriage and the like.

With the preceding observations in hand, you see why the elements (content) of the initial set of support merit some consideration. Indeed, just as misleading clues offer a big obstacle in a treasure hunt, so does the presence of too few or too many clues. I now offer some notions about the use of the set of support strategy, about how the (initial) set of support might be modified, by your reasoning program, as the search for the treasure proceeds.

In the late 1960s, David Luckham (a colleague of mine in the field then called automated theorem proving) made the following intriguing observation. He correctly noted that, when the set of support strategy is in use, far more partitioning occurs at level 1 than at higher levels. Specifically, many of the clauses that would have been deduced at level 1 are not deduced because their parents are in the complement of the initial set of support, which often contributed markedly to the likelihood of success. (Of course, obviously, level 2 is smaller than it would have been because of the missing clauses on level 1 that were not generated.) This partitioning occurs because level 0 is partitioned into the initial set of support and its complement, in OTTER, list(usable).

2.1. Partitioning Enhancement for Tuning the Initial Set of Support List

The following enhancement, which extends the spirit of the set of support strategy to levels 2 and higher in the sense just illustrated, might merit study. With the focus on level-2 clauses, when a clause of level-2 is deduced, it is adjoined to the set of support list if and only if at least one of its parents is a member of the initial set of support. If none of its parents is a member of the initial set of support, and if the clause is to be retained, it is immediately placed on list(usable) (for OTTER), and the newly retained clause is not allowed to initiate paths of reasoning. If the proposed enhancement is available and applied to level 3 and beyond, then, clearly, the levels will not grow so dramatically. One of the side effects of using this enhancement is the increase, in a practical sense, in the use of a breadth-first (level-saturation) approach; indeed, a breadth-first search is far more usable if the size of the levels increases slowly. Of course, the proposed enhancement (if used) increases the likelihood that all paths to reaching the goal are blocked.

2.2. Maximal-Level Partitioning Enhancement for Tuning the Initial Set of Support List

A somewhat related variant of the just-described enhancement gives the user more responsibility and more latitude, and it has a smaller chance of blocking all paths to a solution to the problem under consideration. In the variant, the criterion for being placed on the list(sos) just given is replaced by a user-chosen maximal level. Specifically, at least one of the parents must have level less than or equal to the chosen maximum in order for the newly retained conclusion to be adjoined to the (list)sos). This variant, as well as that previously described, is a restriction strategy. It restricts the program's reasoning in a manner that is consistent with the more familiar set of support strategy, which restricts a program's reasoning by avoiding the deduction of many level-1 clauses that could be deduced if all input clauses were allowed to be considered in all combinations as parents.

2.3. Discarding Enhancement for Tuning the Initial Set of Support List

A more dangerous variant (also a restriction strategy) of either of the two enhancements has the program simply discard any newly deduced clause when the decision is not to adjoin the new item to the set of support list. The danger rests with the total blocking of newly deduced items that are not placed on list(sos), which is indeed quite a restriction to the program's reasoning.

Yes, it is hoped, you will invent other related enhancements.

2.4. Direction Strategy Enhancement for Tuning the Initial Set of Support List

For an enhancement that has the flavor of a direction strategy, the newly retained clause has its weight (priority) adjusted in part based on the nature of its parents. Other variations can be formulated; see the enhancements discussed in this section.

3. Inheritance

With the use of the set of support strategy, you could say that inheritance is present. Indeed, a clause deduced and retained, when this strategy is used, also immediately has support in the sense that it can be chosen to initiate inference-rule applications.

For a discussion of the next enhancement, which I find unlike those discussed in the preceding section, some background will prove valuable.

The likelihood of solving a mystery, answering a question, proving a theorem, or carrying out other activities that rely on careful and logical reasoning often is increased by the deduction of some key fact or key facts. Such deductions are occasionally crucial. In mathematics and in logic, for example, lemmas play a key role, results deduced while seeking to reach the target. If you are skilled, or perhaps fortunate, when you deduce a key lemma, you recognize its importance and quickly rely on its use. So, you might wonder, how can an automated reasoning program (in effect) emulate this effective move?

The method I use is to include in the input the negations of lemmas conjectured to be useful; for OTTER, they are placed in list(passive). I also include in the input resonators that correspond to (the positive form of) each lemma, formulas or equations that closely resemble the lemmas conjectured to be of use,

where the specific variables are treated as indistinguishable. To each such resonator, I assign a small weight, or priority, so that when and if one of the lemmas is deduced, the program will quickly key on the new thought-to-be powerful result. To be explicit, my actions are not designed to precisely emulate the approach a person might take; rather, the goal is to provide another direction strategy to enable the reasoning program to follow one or more promising lines of attack.

Even if an included lemma is not proved, its presence as a resonator can have a marked effect on the program's attack because all variables in a resonator are treated as indistinguishable, ignored other than being recognized as a variable. Therefore, if the program deduces and retains a new conclusion (clause) *A* that is similar in shape to an included resonator *B*, although *A* is not identical to *B*, the newly retained conclusion will be given the same priority as the lemma that prompted the inclusion of the resonator *B*. Clearly, an illustration is in order.

Consider the following resonator *B*, and assume that *B* corresponds to a lemma chosen in phase one of lemma adjunction but not proved; lemma adjunction is fully treated in Chapter 2 of the book *Automated Reasoning and the Discovery of Missing and Elegant Proofs*. (The clause language relies on predicates such as *P*, which here can be interpreted as "provable".)

weight(P(i(i(i(i(x,y),i(z,y)),y),i(i(z,x),z))),2).

If the program deduces and retains the following clause *A*, then *A* will be given high priority for directing the program's reasoning because (treating all variables as indistinguishable) *A* is similar to *B*.

P(i(i(i(i(x,y),i(z,y)),u),i(i(z,x),u))).

The fact that *A* does not correspond to a chosen lemma but, nevertheless, is given high priority illustrates how the inclusion of an unprovable or unproved lemma can affect the program's reasoning. From a practical viewpoint, a focus on *A* may be the key to completing a desired proof. In other words, (as illustrated) you need not choose the precise lemmas that play a vital role.

I have had great success putting the negations of lemmas in list(passive) with the (positive) form included as resonators. Experiments over many years has provided evidence of the power of this approach, whether the goal is finding a shorter proof than in hand, finding a so-called first proof, or simply reducing the CPU time required to complete a given assignment. But what about the child of a newly deduced lemma? Can and should that child be given preference, special consideration? In research conducted by an unaided mathematician or logician, say, the discovery of a lemma is often immediately followed by a line of inquiry that is based on the new discovery. In particular, a child of such a lemma, a deduction in which the new lemma is a parent, is given much consideration.

3.1. Lemma Inheritance Enhancement

The enhancement in focus here captures that intention. Specifically, when a lemma is proved (whose negation is included as a target and whose positive form is included as an item to be given much preference for inference-rule initiation), a child of that lemma would be given the same priority or nearly the same. Further, a grandchild would also be given preference, when and if deduced, but, as expected, less preference than the newly proved lemma. With this enhancement, a reasoning program would be encouraged to pursue, sometimes sharply, a line of reasoning that stems from a newly proved item that the user of the program conjectured to be significant. The given enhancement could be thought of as a lemma-inheritance ability, and reliance on its use gives the program yet another direction strategy to employ.

3.2. Resonator Inheritance Enhancement

In a related way, you could add to a program a resonator-inheritance strategy for also directing the program's reasoning. With this enhancement, as you probably have guessed, the child of an included resonator is given the same or somewhat less priority as are formulas that match the cited resonator. A grandchild inherits the power of the resonator, but with a decrease in its weight or priority. As far as I am aware at the moment, little is now available in the spirit of inheritance.

3.3. Axiom Inheritance Enhancement

I hope by now you have gotten the bug, joined the game, and are ready to ask about other types of inheritance strategies. For example, what about an axiomatic-inheritance strategy? The child of an axiom would be given almost the same power as an axiom is given, and the grandchild somewhat less but more power than it would ordinarily be given. As those familiar with my approach know, in general I prefer axioms to be used to complete applications of inference rules, but not to initiate their use. Indeed, the set of support strategy encourages the user to place axioms outside of the set of support for many, many studies. With a level-saturation (breadth-first) search, children of elements in the set of support are immediately considered before grandchildren, which are immediately considered (in focus) before great-grandchildren, and the like. For many assignments, however, a level-saturation approach is not practical. What could be made available, in the spirit of so-called axiomatic inheritance, is the assigning for inference-rule application initiation a high priority to, say, the child of a member of the (initial) set of support. You see the generalization that might be indeed interesting.

3.4. Hot List Enhancement

Possibly the genesis of the following strategy—a strategy perhaps closer to axiomatic inheritance—is the insightful observation by Branden Fitelson that, when studying a single axiom, the heat should be assigned a high value, perhaps as much as 10. Indeed, an examination of proofs such as Meredith's of his single axiom (for classical propositional calculus) used to derive Lukasiewicz 1 2 3 shows that sequences of steps exist in which each relies on the use of the single axiom. Further, and here is why a high heat value, the second element of the sequence relies on the first, the third upon the second, and the like. The proposed strategy, for directing the reasoning, would adjust the weight of the newly retained clause in accordance with the number of occurrences of certain chosen axioms in its history. For but one example, a child of a pair in condensed detachment with one parent being the single axiom would be given more preference, because of its parentage, than its symbol count would give it. If both parents were the single axiom, even more preference would be given.

3.5. Other Possible Related Enhancements

A second variation would adjust the weight based on the history in the context of how many and which clauses were used from the initial set of support. A third variation would make preference adjustments based on a user-chosen set of preferences for the initial set of support members. For example, the Robbins equation is far more significant than either of associativity or commutativity. Penalties and rewards could be used based on the nature of the immediate parents of a newly retained clause, in the context of the preceding.

Although not precisely in the spirit of inheritance, perhaps the heat of a clause would be used to modify its weight. Ordinarily, the higher the heat, the more preference to be given, but not necessarily. Perhaps, through experimentation in a given study, heat=4 is determined to be more significant than heat=6.

4. Self-Analysis

Of a different nature, in the spirit of learning and self-analysis, the following strategy merits study. You place in the passive list the negations of lemmas, whether or not proved, thought to be of interest or crucial to reaching the goal. Whenever one of those lemmas is proved, have the program pause, adjoin the positive form of it as a resonator, and adjust the weights of the matching clauses currently on the set of support commensurate with the value assigned the new resonator, usually a small one. For a variation, all proof steps of such a lemma would also be adjoined as resonators and adjustments made to the matching clauses on the set of support.

5. Summary and Notes

For total clarity, the goal is *not* to produce a program that is self-sufficient. Rather, the objective is to have a program that can, in effect, frequently discuss with itself how things are going. The program I envision would, therefore, be a far more valuable, automated reasoning assistant than it is now. Of course, as

shown by the splendid achievements being made here in the year 2010, programs such as OTTER are far more powerful than they were but thirty years ago. That dialogue would determine whether progress is being made, which parameters to change, what other actions to take. The program could and would recognize failures and successes, and benefit from identifying such. To save the time that now is not put to use,, for example, during the night as the computer sits idle while it waits for more instructions, is not nearly as crucial as the sharp decrease in time (with the envisioned program) required to reach one goal after another. Indeed, when you can test ideas in hours rather than in weeks, your eagerness and pleasure heightens, and the likelihood of producing a new powerful methodology is also increased greatly. The program I envision would behave as if it is learning, locally, learning from the results that it analyzes as it seeks to find a first proof, discover a shorter proof, or reach some other objective.

In lieu of extensive programming, perhaps one or more of the enhancements could be studied or simulated, to gain some insight into its value, with the use of demodulation, weighting, or some other mechanism. In the spirit of the preceding observation, to determine which elements are best placed in the initial set of support, you could run an experiment in which all input clauses are placed on the list(sos). An examination of the actions of the program might suggest strongly what you would be wise to do. Somewhat related, in my research focusing on classical propositional calculus, I have often in part focused on 68 these cited by Lukasiewicz. Their inclusion, as resonators and or (in negated form) as intermediate targets, has sometimes led me to good choices about values to assign to various parameters. For example, even though proofs of one or more of the 68 had little to do with reaching the main target, such proofs occasionally suggested progress was being made.

As is clear from this notebook, and others found on my website, reliance on an automated reasoning program is most appealing, and the results are most satisfying. I like the fact that, rather than two weeks of thought and unaided experimentation, followed by the conclusion that the approach is not worth pursuing, A good assistant (in the form of a program such as OTTER) permits a huge number of experiments in a very short time.

Until we meet in the next notebook, I wish you excitement and pleasure as you consider what is offered here.