What are the Connections, Inter-relations and Antipathies Between Proof Checking and Automated Theorem Proving?

Deepak Kapur*

There are more commonalities and connections between automated theorem proving and proof checking than are often mentioned. Automatic theorem provers (e.g. *OTTER* and *RRL*) can prove many interesting, nontrivial theorem automatically. For most nontrivial theorems, however, even automatic theorem provers (ATPs) need guidance for finding their proofs. This guidance to ATPs could be in the form of setting knobs/parameters to select appropriate heuristics and inference methods, and/or providing intermediate lemmas that could potentially help in finding a proof. Because of this, automated theorem proving programs are also called *mechanical theorem provers*, *proof assistants*, etc. In this sense, they are similar to proof checkers (PCs).

ATPs and PCs differ in the degree of automation supported by them, and level of granularity of inference steps. As the name suggests, proof checkers are used to check manual proofs. At every proof step, a PC user may have to specify the next inference rule to be applied.

We highlight interrelationships between automated theorem proving and proof checking, discussing possibilities for cooperation between the developers of ATPs and PCs. We review different emphasis in concerns of two different strands of research in mechanical reasoning.

1 Finding Proofs

In an ATP, the emphasis is on finding a proof whereas in a PC, the emphasis shifts to proof checking. It is expected that a user has already done a rough

*e-mail: `kapur@cs.albany.edu`

detailed proof on paper before it approaches a PC. The user interacts with the PC primarily to find any gaps and/or errors in a manual rough proof. In contrast, the objective for using an ATP is to automatically generating a proof, or if that does not seem possible, to get as much help from the ATP in finding a proof. The user is not expected to have a proof already.

Because of different objectives, most PCs implement "small" inference steps, which are used by the user to check the manually obtained hand proof. In contrast, an ATP supports hard-wired large inference steps, typically implemented using smaller primitive inference steps which may or may not be made available to a user. Heuristics play a crucial role in the effectiveness of an ATP. There are many ways to combine primitive inference steps to be used for finding proofs in an ATP, consequently many parameters must be selected and set to an appropriate value. This makes ATPs quite flexible and effective in their use for experts, but difficult for new users.

1.1 Failed Proof Attempts

The mode of proof generation using a PC is quite different than using an ATP. Proof generation is done on a PC under user control, so if a proof attempt does not succeed, the user is likely to know where and why it is failing. This may point to a gap or an error in a manual proof, leading to the user rethinking about the manual proof.

When a proof attempt does not succeed using an ATP, then it is often not easy for a novice user to figure out a reason why the proof attempt might not have succeeded. An expert user of an ATP would look at the transcript, guess whether a lemma is needed or a different proof strategy should be used. A proof is thus generated using "trial and debug" mode, much like a hand proof is obtained or a program is constructed. It is this aspect where there is considerable scope for research and for providing aids to users to help extract useful information from an unsuccessful proof attempt.

ATP builders would have to provide modes in which powerful inference methods supported by an ATP can be invoked under user control, much like a proof checker. Having an ATP run like a proof checker can be useful to attack interesting, nontrivial theorems, where user guidance is necessary. PC developers should not hesitate to adapt automated reasoning techniques commonly used in ATPs so that PC systems become less tedious and more effective to use.

2 Foundations of ATPs and PCs

PC builders have been very concerned about foundational issues related to proof generation and proof finding. Whereas interactive sequent based proof checkers support many inference steps (an inference step for introduction and elimination of every logical connective), a PC supports a small set of inference steps, but provides mechanisms for combining these inference steps to get larger derived inference steps. The main rationale is that soundness and related foundational issues can be more easily ascertained with a small set of primitive inference steps than with a large set of derived big inference steps. Soundness/correctness of combining mechanisms becomes, however, a major burden.

Soundness is a serious concern for ATPs which implement complex heuristics combining primitive inference steps. A small, trivial mistake can generate incorrect proofs thus shaking any confidence in an ATP; one starts wondering whether proofs obtained earlier are correct.

Developers of PCs have emphasized construction of detailed, fine-grained proof objects whereas proof objects are considered a burden by ATP developers, even sometimes a nuisance as generating them is not viewed as being of any use.

3 Cooperative Building of Knowledge Base

The QED project should build on the work of both the ATP and PC groups. One possible approach is to share proofs generated by different ATPs and PCs for building on each other's work. There is a need to develop mechanisms for sharing proofs and permanently storing them in a data base that theorem provers and proof checkers can interface with. A common notation (syntax) for proofs must be developed. How can an ATP or a PC use proofs generated by other ATPs/PCs in its work? This may require developing front-ends for ATPs and PCs in order to use proofs.

4 What is an Acceptable Proof?

Since the first QED workshop, there has been considerable interest among ATP developers to start thinking about proof objects underlying ATPs, how proof objects should be generated, and subsequently used. The Eves group of ORA, Canada, for instance, has successfully demonstrated how proof objects can be generated for Eves, a sophisticated system supporting reasoning by induction, employing decision procedures and other heuristics. McCune has developed a method for generating proof objects from Otter using basic inference steps of binary resolution, factoring on propositional calculus and instantiation. I outlined some preliminary ideas for possibly generating a proof object for a rewrite rule based prover such as *Rewrite Rule Laboratory* (*RRL*).

A consensus is emerging that a proof object should provide justifications for propositional reasoning (i.e., rules for boolean connectives), equality reasoning on ground terms (i.e., reflexivity, symmetry, transitivity, and functional application preserving equality), and universal instantiation. Perhaps some doubts can be raised about justification for the instantiation rule because of the discussion about indefiniteness earlier. There are issues related to instantiating a free variable with a term which has no value or no meaning. There are also issues related to typed variables. Can we instantiate a free typed variable if there is no guarantee that the type denotes an nonempty set? These concerns need to be adequately addressed as they are likely to affect the definition of acceptable proof objects.

There does not seem to be any agreement about a common meta logic regarding proofs by induction. One view is that set theory can serve as an appropriate meta logic, given that induction can be obtained as a derived rule in set theory, for example in ZFC or heavy-duty set theory proposed by McCarthy. This proposal is considered by others as one similar to writing programs in the language of Turing machines or in an assembly language. There is also Mizar view that first order set theory is adequate except that for developing proofs and formalization, it is convenient to have available a second-order feature which is really a syntactic mechanism to provide ability to substitute for meta variables over formulas.

Some of us from programming background believe that we need to agree on some mechanism for introducing structured objects. One proposal in that direction is to adopt Feferman's *Primitive Recursive Arithmetic* (*PRA*) that provides the notation of S-expressions (trees) as a basic data structure along with numbers. Others would like to support a mechanism for defining abstract data types generated by finite set of constructors (something similar to *shell* principle of Boyer and Moore's logic). More discussion is needed on this important issue as inductive reasoning may be critical in many QED applications.

4.1 Objects generated by decision procedures and other algorithms

Many ATPs implement decision procedures for commonly used theories so that a user of these systems is not burdened with having to prove simple obvious facts. Popular theories whose decision procedures are directly encoded into ATPs, are propositional calculus, theory of equality, and linear arithmetic (quantifier-free theory of numbers). Even developers of PCs have recognized the need for supporting such decision procedures, and there is an increasing trend towards providing specialized tactics implementing decision procedures for some theories.

Generating proof objects based on inference steps in a decision procedure is an interesting research issue. The Eves attempt is a good start.

As more capabilities are included in reasoning systems, particularly algorithms from symbolic computation (computer algebra) system, there would be a need to generate proof objects related to such computations also. It appears that for operations such as factorization of polynomials, computing integrals and differentiation, it is perhaps not that difficult to develop a proof object in the form of a certificate that ensures that the result is indeed correct, without having to know how the result is computed. However, this may not be straightforward in other nontrivial computations, such as resultants, Gröbner basis and other elimination techniques. There are new issues raised due to generation of huge objects and the so-called "intermediate expression swell" problem.

4.2 Representational Issues

There are other issued related to computer representation of a proof object. Should it be a linear object, much like a Hilbert style proof, or it should be a directed acyclic graph allowing sharing among parts? Should it be a hierarchical object or a nested object? What additional structure should a proof object have? Should it be a λ term?