On Proving Assistants in the Classroom (and Elsewhere)

Extended Abstract[*]

Wolfgang Schreiner
Wolfgang.Schreiner@risc.uni-linz.ac.at
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

1 Motivation

In classical mathematics education (schools and universities alike) emphasis is given to two particular activities: *calculating* (transforming a given representation of an object to another simpler one) and *solving* (finding objects that satisfy given properties). The third major aspect of mathematics [2], *proving* (reasoning why a property holds for an infinite class of objects), is widely neglected. Moreover, since students typically work in mathematical frameworks previously established by others, the fundamental aspect of *modeling* (finding properties that adequately characterize a problem domain) is mainly left out of consideration.

This is the more disturbing since calculating and solving are exactly those aspects that are nowadays handled best by computers, while modeling and reasoning remain human key qualifications in modern professions: the first task of every non-trivial project is to write a *specification* that describes the desired outcome as precisely as possible; next, this specification is *validated* by some sort of critical analysis; after the specification has been implemented, the result is *verified* with respect to the specification. If these activities are formalized, a specification becomes a mathematical theory, validation means proving consequences of this theory, and verification means proving that the result satisfies the properties demanded by the theory.

One (not the only) example of this kind is *software development*: the formal specification of a computer program describes a binary relation between the program's input state and its output state; this specification can be validated by checking whether the formula holds on desired input/output examples (as a consequence, the specification is not void) and does not hold on undesired ones (as a consequence, the specification is not trivial). Finally, (a model of) the program can be verified by deriving from its source code formal conditions that, if proved as true, ensure that any state transition that can be performed by the program is indeed allowed by the specification.

Program specifications can serve as a rich source of examples: writing, validating, and analyzing specifications, as well as proving that they are met by implementations, is of application-independent *general value*. In particular, just by using well-known

[*]Submission for Session ConvMathAssist

objects with generally well-understood mathematical theories (natural numbers and finite sequences suffice), one can already construct interesting specifications of intuitive problems and can derive from simple programs corresponding verification conditions. Moreover, even basic problem specifications require the full expressiveness of the *language of predicate logic*; if the object theory is simple, any problems and errors when formulating and arguing about properties arise from true logical errors.

Actually, to get proficient with the practical use of the language of predicate logic is one of the most important goals of such a training because it is a prerequisite for *the precise formulation of properties and relationships*. This language is hardly taught in school and only rudimentary in university education; as a consequence, even master students of mathematics and computer science make fundamental mistakes when expressing intuitive statements in a formal way. Since language shapes thought, this is not only a superficial blemish but represents a fundamental problem which constantly hampers thinking/communicating/arguing about complex facts and relationships.

2 Tool Support

So, if one accepts the need for training in the language of logic as an important and integral part of mathematics education, which kind of computational tools related to logic can help to achieve these goals?

Visualization/animation tools [8] may help to grasp the interpretation of logic formulas over graphically illustrated domains of objects; however, they give little aid to understand the general principles of formal reasoning. *Proof checkers* [10] allow to verify the correctness of proofs; however, they are of little help in the construction of such proofs (which furthermore have to be elaborated in a tedious level of detail). *Automated theorem provers* [3] attempt to automatically construct proofs of given properties. However, if proof attempts *fails*, typically the only chance for the user is to re-adjust the general proving strategy and restart the prover. While the user may thus be trained in devising general proving heuristics respectively adjusting them to concrete proofs, this relies on the fact that the user has already fundamental proving skills. On the other side, if the proof *succeeds*, the only way that a student may learn from this result is by studying the individual steps of the automatically constructed proof. This is a passive act of consumption rather than an active act of construction, which diminishes its educational effects.

Interactive proving assistants [4, 5, 7] support the user in the construction of a proof by displaying a proof in a structured form and presenting in every proving situation those inference rules that are currently applicable. The main task of the user to select appropriate rules and provide critical additional information (in particular terms by which universally quantified assumptions respectively existentially quantified goals shall be instantiated) respectively select/construct appropriate lemmas which introduce new information (and which have to be validated in separate proofs); the tool then takes care of the correct application of the rule such that final proofs are guaranteed to be correct. By the active contribution of a human user with intuitive insight, proofs of bigger complexity can be handled than by purely automatic provers; furthermore, using such a tool demands active participation, which may greatly enhance the educational effect. On the negative side, the practical effectiveness of such tools depends to a good deal on the adequacy of the user interface with respect to proof presentation/navigation and on the concrete mode of interaction. Furthermore, depending on granularity of inference steps provided by the system, proofs may become tedious; especially dealing with

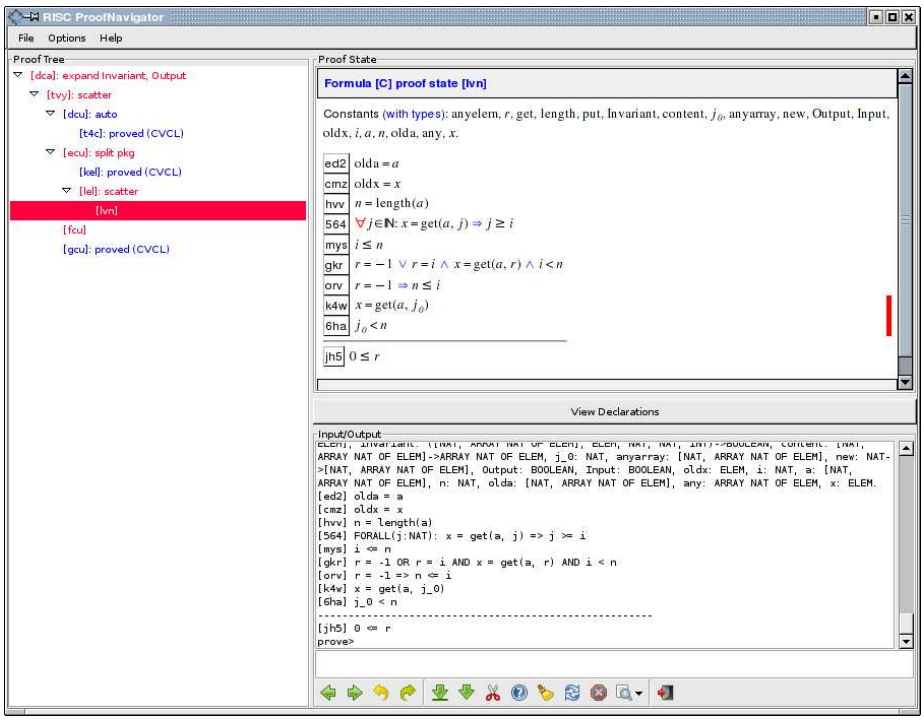

Figure 1: The RISC ProofNavigator in Action

arithmetic properties may become a nuisance, if they are reduced to basic reasoning in arithmetic theories.

Taking all of these trade-offs into account, we believe that interactive proving assistants have the most prospective for use in educational scenarios where the goal is to train the practical use of the language of logic with respect to formal modeling and reasoning; however, success does not come for free but great care has to be taken to diminish the problematic aspects.

3 The RISC ProofNavigator

Since 2004, we have taught every year at the Johannes Kepler University Linz a mandatory course on "Formal Methods in Software Development" for master students of computer mathematics and (since 2007) software engineering; originally, we introduced in this course students to the proving assistant PVS [4]. Based on our experience, we subsequently developed the *RISC ProofNavigator* [6] as a tool that we consider more suitable for (among others) educational scenarios; this software is since 2007 also used in a similar course at the Upper Austria University of Applied Sciences in Hagenberg. A screenshot of the RISC ProofNavigator is depicted in Figure 1. In the following, we quickly outline its use; a more comprehensive description can be found in [7].

The RISC ProofNavigator is based on a typed higher order variant of predicate logic; it reads from a text file a sequence of definitions (of types, functions, predicates, and formulas) in a plain text format; these are pretty-printed in a screen area in the

classical mathematical format. For each formula, the current status of its (possibly previously constructed) proof is displayed. The user may select a completed proof for display/replay or an open/incomplete proof for further processing.

A proof is depicted as a tree of proof situations ("states") which are either marked as red (the situation respectively some sub-situation is still open) or blue (the situation is closed, i.e. the corresponding proof is completed). The user can freely navigate among (especially open) situations by mouse clicks or by a pressing a buttons at the bottom of the window; the currently selected situation is depicted in a pretty-printed format in the "Proof State" area and in the linear format in the "Input/Output" area. Proof commands may be applied to the current situation by pressing a button at the bottom, by selection from a menu linked to a formula label, or by selection from a general menu. If the command requires additional information, it is displayed in the input line with placeholders that can be filled by the user. By application of the command, the proof tree is expanded at the current situation with some/all subnodes possibly closed. If all nodes are closed, the proof is completed and the tree turns blue. A proof (be it complete or incomplete) may be saved on file for later replay.

In the design of the frontend, much attention was paid to those details that are important for practical use (but often neglected): first, proofs are displayed as trees rather than as unorganized lists of open proof situations; this allows the user to keep much easier track of the overall state of the proof. Formulas are pretty-printed in a classical format with nice mathematical fonts; large formulas are appropriately formatted and indented across multiple lines such that their logical structure is visually preserved. In proof states, both the pretty-printed as the linear textual presentation is presented; the linear presentation is the one used for user input (e.g. when providing witness terms). While linear text input seems initially more cumbersome than some form of WYSIWYG input, it is after some initial training actually much easier and faster to use. If a new proof situation is generated, the difference to its parent situation is depicted by red bars on the right side of the proof state area; this allows to quickly grasp the effect of the applied proof command.

As for user interaction, the six most often applicable commands can be triggered by pressing buttons with intuitive icons; if a command depends on a particular assumption or goal, it can be selected from the menu that pops up by moving the mouse cursor over the label of the corresponding formula. Only in rare situations a command may be needed that can be selected from a general menu (in total, there are less than 30 commands); in any case, the user may always type the name of the command rather than looking it up in a menu. Since formula labels are automatically generated by hashing the text of a formula, previously recorded applications of proof commands may generally remain applicable, even if the user changes details in the definitions of entities; this allows repeated iterations of modeling and reasoning steps without completely invalidating all previously constructed proofs.

4 Proving in the RISC ProofNavigator

A major design decision for a proof assistant concerns the granularity of proof steps, how to simplify proof situations, and how to deal with arithmetic. As a core component of the RISC ProofNavigator, we employ the software CVCL (Cooperating Validity Checker Lite [1]). CVCL is a representative of the class of SMT (Satisfiability Modulo Theories [9]) solvers which has emerged in the last decade. SMT solvers have revolutionized the area of program verification since they implement efficient and fully

automatic decision procedures for certain combinations of logical theories including uninterpreted functions with equality (i.e. term reasoning) and linear arithmetic over the integers. In other words, CVCL is able to decide propositional logic formulas involving axiomatized functions/predicates, integers with addition and subtraction, as well as equalities and inequalities; it may also transform such formulas to a simpler (not necessarily canonical) form.

As a consequence, the core of the RISC ProofNavigator does not at all deal with reasoning on terms and numbers but focuses on the logical structure of a proof; by default, after every application of a proof command, the resulting proof situation is forwarded to the SMT solver which may be able to close it. If not, every formula is presented to the solver for simplification in the context of the current proof situation; from this e.g. in a proof situation with two assumptions of form A and $A \Rightarrow B$, the second formula is immediately simplified to B (i.e. the rule "modus ponens" is automatically applied). Since in this way a lot of "low-level reasoning" is fully automated, proofs become greatly simplified; we consider this as a crucial advantage of our system. Nevertheless, we found the results sometimes too surprising for class-room scenarios ("too much simplification in one step"); we therefore have introduced later an option to selectively switch off the automatic form of simplification; then the user may explicitly decide to apply simplification to some or all formulas of a proof situation.

Another problem with rewriting represents the presentation of disjunctions and implications: logically, e.g. the three formulas $\neg A \vee B \vee C$, $A \Rightarrow B \vee C$, $A \wedge \neg C \Rightarrow B$ are equivalent but represent different human intuitions. Since CVCL has a preference to replace implications by disjunctions, we have integrated a post-processor to rewrite formulas to a form which we consider more natural (as a guiding principle, we attempt to minimize the number of negations, so among the three alternatives above, we chose the second one); however, also this is a less than perfect solution since it also changes the format of the formulas entered by the user. In general, it is still an open question how to allow automatic formula simplification but presents a formula such that is intuitive to the user.

A related problem concerns the presentation of proof situations. Rather than proving $A, \neg B \vdash C$ (i.e. proving from assumptions A and $\neg B$ goal C), we might prove $A, \neg C \vdash B$ or $A, \neg B, \neg C \vdash$ false where the last represents a "counterexample" proof showing that the conjunction of assumptions represents a contradiction. While logically all are the same, they represent different human intuitions; in particular, it is extremely annoying to the user, if a goal $\exists x : A_x$ becomes an assumption $\forall x : \neg A_x$ (which happens if the system decides to transform the goal into a negated assumption). We therefore decided never to swap the role of a formula (from goal to assumption or vice versa) but provide a `swap` command that allows the user to change his view by swapping a goal with a particular assumption (thus negating both). Furthermore, while the underlying logical calculus allows more than one goal ($\ldots \vdash A, B$ means to prove A or B), all our commands generate only one goal formula (respectively none, if the proof is a counterexample proof).

As for the proof commands implemented by the core of the RISC ProofNavigator itself, we decided not to provide individual low-level inference rules rather than "generic" commands and combinations of such commands. On the lowest level, we have e.g. a `flatten` command, which applies that rule that is appropriate when considering the outermost logical connective of an assumption or goal, provided its result is a single proof situation (e.g. an assumption $A \wedge B$ is transformed to two assumptions A, B); likewise a command `split` splits a proof situation in two considering the outermost logical connective of a formula (e.g. a goal $A \wedge B$ generates two proof situations,

one with goal A and one with goal B). Likewise, a command `skolem` removes the quantifier from an existential assumption or universal goal by introducing a "Skolem constant". However, these low-level commands are only selectable from a generic menu, because in practice one of the "meta-rule" commands is invoked which applies a combination of such rules. The most aggressive command is `scatter` which applies as many `flatten` and `split` steps as possible before depicting the result (a new subtree); less aggressive is `decompose` which only applies `flatten` steps and thus results in a single new proof situation (no branching occurs).

In order to automate simple proofs depending on the construction of witness terms, the `auto` command implements a simple heuristics where a number of automatic instantiations of quantified variables are generated from those subterms that occur in the current proof situation and that have appropriate type. The resulting proof situation is passed to the SMT solver which, if we are lucky, closes the situation; otherwise, the command has no effect (i.e. the generated proof situation is discarded). Many basic proof situations can be indeed conveniently handled in this way; if one, however, is interested in the "right" instantiation (which is not directly visible from the many generated ones), the user has to construct it manually. The `autostar` command applies automatic proof situations to all open "sibling" situations (which is successful in many verifications).

While thus some automation was implemented to make the low-level parts of a proof less painful, this automation was carefully selected such that executing a command (pressing a button) has a predictable behavior; in particular it terminates after a couple of seconds (perhaps with no success). No elaborated automated proof search is attempted since this has unpredictable behavior and is in realistic situations rarely helpful.

Moreover, it is often the case that users attempt to prove *wrong* formulas where no proof is possible at all; in such situations it is important to get intuition why the proof does not work (is the formula wrong? is the proof strategy inadequate?) and take appropriate measures. For this purpose, the RISC ProofNavigator provides the command `counterexample` which asks the CVCL subsystem for an interpretation of the constants in the current proof state which may invalidate the goal; the result is a list of equalities which is displayed to the user and may give some hint on the problem. However, this is only a small support; in particular, if the software has already simplified a proof state too much (the proof of a wrong goal may reduce to $\ldots \vdash \mathrm{false}$) it is necessary to get back to earlier proof situations which yield more insight. More research is needed on how to give appropriate hints on the underlying reason why a proof might not work.

5 Experience and Conclusions

We have used the RISC ProofNavigator for three years in university courses on formal methods (actually only four weeks are dedicated to the basic principles of program verification and the use of this tool for performing the core proofs). The basic assumption of these courses is that students are already familiar with logic from their bachelor studies and only need a quick reminder on the basic principles of formal reasoning; however, this has turned out to be unrealistic, since many students lack this knowledge. Our following observations thus are based on classes with heterogeneous backgrounds where appropriate compromises had to be made to overcome the gap between the official goals and the factual situation.

- The use of a proving assistant indeed helps to increase the quality of proofs (compared to assignments with paper and pencil proofs where the results are in many cases hardly proofs at all); furthermore students sensually perceive the difference between a proof sketch and a "real" proof by a proof tree turning from red to blue (which represents a concrete achievement with a corresponding satisfaction).

- In the relatively short time available (about 16 teaching units), by developing and presenting concrete example proofs in the class room (including the explanation of overall proving strategies and practical on-line demonstration of the software), the majority of students becomes able to successfully produce structurally "similar" proofs; various students actually seem to enjoy the challenge of working with the proof assistant and guiding it to produce the desired result.

- Nevertheless, there is the persistent danger that a student gets tired or bored and switches from "thinking mode" to "button pressing mode" where (like in a computer adventure game) random actions are taken in the hope of producing a successful result. We have seen proofs with more than a hundred command applications where less than a dozen would have sufficed; these were apparently produced in such a mode. Students that lack interest in finding out whether/why something is true but just want to get the exercise done, also do not get enthusiastic just by using a tool.

- Especially in the beginning, it must be strongly recommended to restrict the use of the proof commands to the "low-level" commands (with no automatic repeated application of decomposition rules and no automatic quantifier instantiation) to understand the individual reasoning steps. Only later when this basic understanding is achieved, the higher level commands may be applied.

- The real challenge is not proving something that is correct but finding out what is wrong with a proposition whose proof attempt fails. Here even bright students have trouble on finding out whether the problem is "just" an inadequate proof strategy or whether the statement is indeed not correct (which means in program verification that the specification may not have the intended meaning, that the program may not meet the specification, or that a loop invariant may be too strong or too weak).

With respect to further extensions/redesigns of the software, it might be wise to reconsider the level to which automated simplification of proof situations by the underlying SMT solver is applied; while desired on the level of atomic formulas and on certain logical rules (in particular, the automated application of modus ponens and similar syllogisms is extremely comfortable), the structural modification of formulas (with subsequent heuristic rewriting to a more suitable form) is often undesired. Furthermore, for educational purposes (and bug-finding) it might be wise to have a possibility to display the various intermediate steps in the higher-level proof commands.

As for arithmetic reasoning, linear arithmetic is effectively supported, but the system completely fails to deal with any form of non-linear formulas (even simple equalities like $a(b+1) = ab+a$ cannot be proved); instead lemmas for the use of the multiplication symbol have to be used and separately proved by induction. Here the use of semi-decision procedures (based e.g. on the capabilities of computer algebra software) would be very helpful.

In general we believe that, by the integration of automated rule-based provers with interactive humane assistance and (semi-)automatic decision procedures for basic theories, the effective use of formal logic in practical scenarios can be demonstrated, in the classroom and elsewhere.

References

[1] Clark Barrett and Sergey Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker. In *Computer Aided Verification: 16th International Conference, CAV 2004, Boston, MA, USA, July 13–17, 2004*, volume 3114 of *LNCS*, pages 515–518. Springer, 2004.

[2] Bruno Buchberger. Computing, Solving, Proving: A Survey on the Theorema Project (Abstract). In Jürgen Dix et al., editors, *Logic Programming and Nonmonotonic Reasoning, 4th International Conference, LPNMR'97, Dagstuhl Castle, Germany, July 28-31, 1997, Proceedings*, volume 1265 of *Lecture Notes in Computer Science*, pages 220–221. Springer, 1997.

[3] Bruno Buchberger et al. A Survey of the Theorema project. In Wolfgang Küchlin, editor, *ISSAC'97 International Symposium on Symbolic and Algebraic Computation*, pages 384–391, Maui, Hawaii, July 21–23, 1997. ACM Press, New York.

[4] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 14–18, 1992. Springer.

[5] Florina Piroi and Temur Kutsia. The Theorema Environment for Interactive Proof Development. In G. Sutcliffe and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning. Proceedings of the 12th International Conference, LPAR'05*, volume 3835 of *Lecture Notes in Artificial Intelligence*, pages 261–275. Springer Verlag, 2005.

[6] The RISC ProofNavigator, 2009. Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, http://www.risc.uni-linz.ac.at/research/formal/software/ProofNavigator.

[7] Wolfgang Schreiner. The RISC ProofNavigator: A Proving Assistant for Program Verification in the Classroom. *Formal Aspects of Computing*, April 2008. DOI 10.1007/s00165-008-0069-4.

[8] David Schwingenschlögl. Visualisierung von prädikatenlogischen Auswertungen unter Berücksichtigung lernfördernder Aspekte (Visualization of Predicate Logic Evaluations Consderinge Educational Aspects, in German). Master's thesis, Engineering for Computer-Based Learning (CBL), Upper Austria University of Applied Sciences, Hagenberg, 2006.

[9] SMT-LIB — The Satisfiability Modulo Theories Library, 2009. University of Iowa, Iowa City, IA, http://combination.cs.uiowa.edu/smtlib.

[10] A. Trybulec and P. Rudnicki. Using Mizar in Computer Aided Instruction of Mathematics. In *Norvegian-French Conference of CAI in Mathematics*, Oslo, Norway, 1993.