On Proving Assistants in the Classroom (and Elsewhere)

Wolfgang Schreiner
Wolfgang.Schreiner@risc.uni-linz.ac.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
http://www.risc.uni-linz.ac.at

1. The Role of Reasoning

2. The RISC ProofNavigator

3. Experience and Conclusions

Mathematics Education

Various kinds of mathematical activities.

- Calculating
  - Transforming a given representation of an object to a simpler one.

$$(x+y)^2 \rightsquigarrow x^2 + 2xy + y^2$$

- Solving
  - Finding objects that satisfy given properties.

$$x^2 - 5x + 6 = 0 \rightsquigarrow x = 2 \vee x = 3$$

- Proving
  - Reasoning whether a property holds for an infinite class of objects.

$$\forall x \in \mathbb{R} : x \geq 0 \Rightarrow \exists y \in \mathbb{R} : x = y^2 \rightsquigarrow \text{true}$$

- Modeling
  - Finding properties that adequately characterize a problem domain.

Traditionally, mathematics education has focused on the first two items.

Real Life Today

- Calculating and Solving
  - Essential competence of computers.
- Modeling and Reasoning
  - Essential competence of humans.
- Typical Project Phases
  - Write a specification that describes desired results.

    Formally: *develop a mathematical theory.*
  - Validate the specification by a critical analysis.

    Formally: *prove theorems in the theory.*
  - Verify the project results with respect to the specification.

    Formally: *prove that objects satisfy theorems.*

Modeling and reasoning (rather than calculating and solving) are necessary key qualifications for modern professions.

Example: Software Development

- Write a software specification.

  Formally: *A relation between a program's input and its output.*

  $$R(x,y) :\Leftrightarrow I(x) \Rightarrow O(x,y)$$

- Validate the specification by a critical analysis.

  Formally: *Prove that the relation holds for some desired outputs and does not hold for some undesired ones.*

  $$R(a,b_0), \neg R(a,b_1)$$

- Verify the project results with respect to the specification.

  Formally: *prove that, for every input, the output computed by the program satisfies the relation.*

  $$\forall x : R(x, F(x))$$

Program specifications can serve as a rich source of examples for mathematical modeling and reasoning.

Example: A Program Specification

Given an array a with elements from T, a position p in a, and a length l, return the array b derived from a by removing $a[p], \ldots, a[p+l]$.

- Input: $a \in T^*$, $p \in \mathbb{N}$, $l \in \mathbb{N}$
- Input condition:

$$p + l \leq \text{length}(a)$$

- Output: $b \in T^*$
- Output condition:

$$\begin{array}{l} \textbf{let } n = \text{length}(a) \textbf{ in} \\ \text{length}(b) = n - l \wedge \\ (\forall i \in \mathbb{N} : i < p \Rightarrow b[i] = a[i]) \wedge \\ (\forall i \in \mathbb{N} : p \leq i < n - l \Rightarrow b[i] = a[i+l]) \end{array}$$

Mathematical theory:

$$T^* := \bigcup_{i \in \mathbb{N}} T^i, T^i := \mathbb{N}_i \rightarrow T, \mathbb{N}_i := \{n \in \mathbb{N} : n < i\}$$
$$\text{length} : T^* \rightarrow \mathbb{N}, \text{length}(a) = \textbf{such } i \in \mathbb{N} : a \in T^i$$

The Language of Predicate Logic

For modeling and reasoning, one needs a precise language.

- The language of predicate logic
  - Atomic propositions, connectives, quantifiers.
- Indispensable tool for understanding statements.
  - Precise description of complex properties and relationships.
  - Framework for thinking, communicating, arguing.
- Hardly taught in school, only rudimentary at universities.
  - Hampers communication a lot.

One important goal of mathematical education is (should be) to train the practical use of this language.

Tool Support

- Visualization/animation tools
  - Help to grasp formula interpretations, not to understand reasoning.
- Proof checkers
  - Help to verify correctness of proofs, not to construct such proofs.
- Automated theorem provers
  - Attempt to automatically construct proofs by automatic strategy.
  - If fails, proof may be restarted with a modified strategy.
  - If successful, proof may be studied
    - A passive act of consumption, not an active act of construction.
- Interactive proving assistants
  - Combination of user interactions and automatic methods.
    - Visualization of a (partial) proof in a structured form.
  - User selects appropriate strategy that is executed by assistant.
    - User may inject *critical insight:* instantiate existential goals or universal assumptions, apply lemmas, etc.
  - Low-level reasoning steps may be completely automated.
    - *SMT (satisfiability modulo theory solvers):*

Proving Assistants

Target: (education in) computer-supported program verification.

- Personal evaluation of several proving assistants (2004/2005).
  - For classroom use as well as for real verifications.
  - Test cases derived from verifications of sequential programs and concurrent systems (from small proofs to rather large ones).
- Frequently more difficult to use than expected.
  - Steep learning curve.
  - Poor usability respectively "look and feel".
- Frequently less helpful than expected.
  - Too little focus on solving simple tasks (become complicated).
  - Too much focus on solving complex tasks (tend to fail).
- Personal favorite: PVS.
  - Practical success was achieved with limited efforts.
  - Also larger verifications became manageable.

Evaluation yielded some insights on key aspects of proving assistants.

Key Aspects of Proving Assistants

- Convenient navigation in proof trees.
  - User gets easily lost in large proofs.
- Aggressive simplification and pretty presentation of proof states.
  - User quickly loses intuition about interpretation of proof situation.
- Automation in dealing with arithmetic.
  - Subtype relationship between integers and reals is helpful.
- Proof construction by combination of
  - Semi-automatic proof decomposition,
    - $\forall$-introduction, $\exists$-elimination, $\wedge$-introduction, etc.
  - Critical steps performed by user,
    - $\forall$-elimination, $\exists$-introduction, case distinction, etc.
  - (Semi-)decision procedures for ground theories.
    - Uninterpreted function symbols, linear arithmetic, etc.
- Proof stability under changes of predicate definitions.
  - If formula positions change, references to positions break.

1. The Role of Reasoning

2. The RISC ProofNavigator

3. Experience and Conclusions

The RISC ProofNavigator

http://www.risc.uni-linz.ac.at/research/formal/software/ProofNavigator

- A proof assistant developed at RISC.
  - Employs the SMT solver CVC Lite (CVCL).
  - Targeted for education in program reasoning.
- Focus on practical aspects of proving.

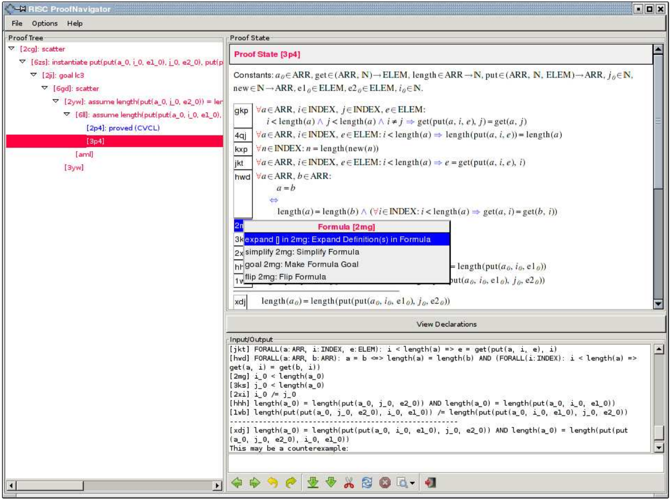
  Rather than on theoretical elegance.

  - Low-level reasoning completely delegated to SMT solver.
    - Equalities, uninterpreted functions, linear arithmetic, ...
  - High-level work made as comfortable as possible.
    - Mainly application of pre-selected proof decomposition strategies.
  - Graphical user interface with convenient interaction possibilities.
- Component of a program exploration environment.

  The RISC ProgramExplorer (under development).

The user deals with the predicate-logic structure of a proof only; equality/inequality reasoning is performed fully automatically.

The RISC ProofNavigator



Using the Software

- Develop a theory.
  - Text file with declarations of types, constants, functions, predicates.
  - Axioms (propositions assumed true) and formulas (to be proved).
- Load the theory.
  - File is read; declarations are parsed and type-checked.
  - Type-checking conditions are generated and proved.
- Prove the formulas in the theory.
  - Human-guided top-down elaboration of proof tree.
  - Steps are recorded for later replay of proof.
  - Proof status is recorded as "open" or "completed".
- Modify theory and repeat above steps.
  - Software maintains dependencies of declarations and proofs.
  - Proofs whose dependencies have changed are tagged as "untrusted".

Exercise in the mathematical aspects of modeling and reasoning.

Proving a Formula

- Proof of formula F is represented as a tree.
  - Each tree node denotes a proof state (*goal*).
    - Logical sequent:
      $A_1, A_2, \ldots \vdash B_1, B_2, \ldots$.
    - Interpretation:
      $(A_1 \wedge A_2 \wedge \ldots) \Rightarrow (B_1 \vee B_2 \vee \ldots)$
  - Initially single node $Axioms \vdash F$.

Constants: $x_0 \in S_0, \ldots$

$[L_1]$	A_1
$\ldots$	
$[L_n]$	A_n
$[L_{n+1}]$	B_1
$\ldots$	
$[L_{n+m}]$	B_m

- The tree must be expanded to completion.
  - Every leaf must denote an obviously valid formula.
    - Some A_i is false or some B_j is true.
- A proof step consists of the application of a proving rule to a goal.
  - Either the goal is recognized as true.
  - Or the goal becomes the parent of a number of children (subgoals).

    The conjunction of the subgoals implies the parent goal.

An Open Proof Tree

Proof Tree

- [tca]: induction n in byu
  - [dbj]: proved (CVCL)
  - [ebj]

Formula [S] proof state [dbj]

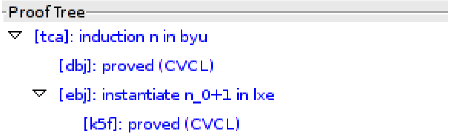
Constants (with types): sum.

lxe	$\forall n \in \mathbb{N}: n > 0 \Rightarrow \text{sum}(n) = n + \text{sum}(n-1)$
d3i	$\text{sum}(0) = 0$
nfq	$\text{sum}(0) = \frac{(0+1)\cdot 0}{2}$

Parent: [tca]

Closed goals are indicated in blue; goals that are open (or have open subgoals) are indicated in red. The red bar denotes the "current" goal.

A Completed Proof Tree



The visual representation of the complete proof structure; by clicking on a node, the corresponding proof state is displayed.

Navigation Commands

Various buttons support navigation in a proof tree.

- : `prev`
  - Go to previous open state in proof tree.
- : `next`
  - Go to next open state in proof tree.
- : `undo`
  - Undo the proof command that was issued in the parent of the current state; this discards the whole proof tree rooted in the parent.
- : `redo`
  - Redo the proof command that was previously issued in the current state but later undone; this restores the discarded proof tree.

Single click on a node in the proof tree displays the corresponding state; double click makes this state the current one.

Proving Commands

The most important proving commands can be also triggered by buttons.

- (`scatter`)
  - Recursively applies decomposition rules to the current proof state and to all generated child states; attempts to close the generated states by the application of a validity checker.
- (`decompose`)
  - Like `scatter` but generates a single child state only (no branching).
- (`split`)
  - Splits current state into multiple children states by applying rule to current goal formula (or a selected formula).
- (`auto`)
  - Attempts to close current state by instantiation of quantified formulas.
- (`autostar`)
  - Attempts to close current state and its siblings by instantiation.

Less frequently used commands can be selected from the menus.

Proving Strategies

- Initially: semi-automatic proof decomposition.
  - `expand` expands constant, function, and predicate definitions.
  - `scatter` aggressively decomposes a proof into subproofs.
  - `decompose` simplifies a proof state without branching.
  - `induction` for proofs over the natural numbers.
- Later: critical hints given by user.
  - `assume` and `case` cut proof states by conditions.
  - `instantiate` provide specific formula instantiations.
- Finally: simple proof states are closed by SMT solver.
  - `auto` and `autostar` may help to close formulas by the heuristic instantiation of quantified formulas.

Appropriate combination of semi-automatic proof decomposition, critical hints given by the user, and the application of an SMT solver is crucial.

Proving Strategies

- Initially: semi-automatic proof decomposition.
  - `expand` expands constant, function, and predicate definitions.
  - `scatter` aggressively decomposes a proof into subproofs.
  - `decompose` simplifies a proof state without branching.
  - `induction` for proofs over the natural numbers.
- Later: critical hints given by user.
  - `assume` and `case` cut proof states by conditions.
  - `instantiate` provide specific formula instantiations.
- Finally: simple proof states are closed by SMT solver.
  - `auto` and `autostar` may help to close formulas by the heuristic instantiation of quantified formulas.

Appropriate combination of semi-automatic proof decomposition, critical hints given by the user, and the application of an SMT solver is crucial.

Example: Verification of Linear Search

0: $\{Input\}$
1: $m := a[0]$
2: $i := 1$
3: $\{Invariant\}$
4: **while** $i < n$ **do**
5:     **if** $a[i] < m$ **then**
6:         $m := a[i]$
7:     $i := i + 1$
8: $\{Output\}$

execution $0 \to 1 \to 2 \to 3$
$V1 \equiv Input \wedge m = a[0] \wedge i = 1 \Rightarrow Inv(m, i)$

execution $3 \to 4(\text{true}) \to 5(\text{true}) \to 6 \to 7 \to 3$
$V2a \equiv Inv(m, i) \wedge i < n \wedge a[i] < m \wedge m_0 = a[i] \wedge i_0 = i + 1 \Rightarrow Inv(m_0, i_0)$

execution $3 \to 4(\text{true}) \to 5(\text{false}) \to 7 \to 3$
$V2b \equiv Inv(m, i) \wedge i < n \wedge a[i] \not< m \wedge i_0 = i + 1 \Rightarrow Inv(m, i_0)$

execution $3 \to 4(\text{false}) \to 8$
$V3 \equiv Inv(m, i) \wedge i \not< n \Rightarrow Output$

Verification conditions correspond to paths in program.

Verification of Linear Search

$$Input \equiv n > 0 \wedge a = olda \wedge n = oldn$$

$$\begin{aligned}
Output \equiv\ & a = olda \wedge n = oldn \wedge \\
& (\forall i \in \mathbb{N} : i < n \Rightarrow m \leq a[i]) \wedge \\
& (\exists i \in \mathbb{N} : i < n \wedge m = a[i])
\end{aligned}$$

$$\begin{aligned}
Invariant(m, i) \equiv\ & \\
& n > 0 \wedge a = olda \wedge n = oldn \wedge \\
& 1 \leq i \leq n \wedge \\
& (\forall j \in \mathbb{N} : j < i \Rightarrow m \leq a[j]) \wedge \\
& (\exists j \in \mathbb{N} : j < i \wedge m = a[j])
\end{aligned}$$

Specification and invariant have to be provided by programmer.

The ProofNavigator Theory

```
a: ARRAY INT OF INT; olda: ARRAY INT OF INT;
n: INT; oldn: INT; m: INT; m_0: INT; i: INT; i_0: INT;

Input: BOOLEAN =
  a = olda AND n = oldn AND n > 0;

Output: BOOLEAN =
  a = olda AND n = oldn AND n > 0 AND
  (FORALL(i: INT): 0 <= i AND i < n => m <= a[i]) AND
  (EXISTS(i: INT): 0 <= i AND i < n AND m = a[i]);

Invariant: (INT, INT) -> BOOLEAN =
  LAMBDA(m: INT, i: INT):
    a = olda AND n = oldn AND n > 0 AND 1 <= i AND i <= n AND
    (FORALL(j: INT): 0 <= j AND j < i => m <= a[j]) AND
    (EXISTS(j: INT): 0 <= j AND j < i AND m = a[j]);

V1: FORMULA
  Input AND m = a[0] AND i = 1 => Invariant(m, i);
V2_a: FORMULA
  Invariant(m, i) AND i < n AND a[i] < m AND m_0 = a[i] AND i_0 = i+1 =>
    Invariant(m_0, i_0);
V2_b: FORMULA
q  Invariant(m, i) AND i < n AND NOT(a[i] < m) AND i_0 = i+1 =>
    Invariant(m, i_0);
V3: FORMULA
  Invariant(m, i) AND NOT(i < n) => Output;
```

The RISC ProofNavigator

V1:
- [2hg]: expand Input, Invariant
  - [6ko]: scatter
    - [21d]: proved (CVCL)
    - [31d]: proved (CVCL)
    - [41d]: auto
      - [nej]: proved (CVCL)

V3:
- [4hg]: expand Invariant, Output
  - [nx5]: scatter
    - [4pd]: proved (CVCL)
    - [5pd]: auto
      - [udv]: proved (CVCL)

V2a:
- [2hg]: expand Input, Invariant
  - [6ko]: scatter
    - [21d]: proved (CVCL)
    - [31d]: proved (CVCL)
    - [41d]: auto
      - [nej]: proved (CVCL)

V2b:
- [hqk]: expand Invariant
  - [thu]: scatter
    - [hfa]: proved (CVCL)
    - [ifa]: proved (CVCL)
    - [jfa]: auto
      - [b41]: proved (CVCL)
    - [kfa]: auto
      - [i3q]: proved (CVCL)

Expanding definitions, decomposing proofs, instantiating quantifiers.

1. The Role of Reasoning

2. The RISC ProofNavigator

3. **Experience and Conclusions**

Experience

"FM in Software Development" at the JKU Linz and FH Hagenberg.

- Courses for MSc programs.
  - About 16 lecture units dedicated to program verification by proving.
  - Students have BSc and should be already familiar with logic.
  - Not all are: variety of backgrounds demands compromises.
- Quality of proofs has considerably increased.
  - Paper-and-pencil proofs were rarely proofs at all.
  - Difference between a proof attempt and a real proof is perceived.
    - Proof tree turns from red to blue.
    - Concrete achievement with corresponding satisfaction.
- Majority becomes enabled to perform moderately complex proofs.
  - Structurally similar to those elaborated in the class room.
  - Some students seem to enjoy the challenge and indeed like to work with the assistant.

Experience

- Tired/bored students switch to "button pressing" mode.
  - Stop to think and perform random actions to get work done (like playing a computer adventure).
    - Proofs with about 100 command applications were submitted (less than a dozen would have sufficed).
  - If a student is not interested in finding out whether something is true or not, using a tool does not change the attitude.
- Initially restrict capabilities of proving assistant.
  - First only allow low-level commands to understand individual reasoning steps.
  - Only later high-level decomposition rules and automatic quantifier instantiation may be used.
- Real challenge is finding out why a proof attempt fails.
  - Is the proof strategy inadequate?
  - Does the program not meet its specification?
  - Does the specification not have the intended meaning?
  - Is the loop invariant to strong or too weak?

Conclusions

The software is limited in various aspects.

- Unexpected structural modifications of formulas by SMT solver.
  - However, automatic simplification of atomic formulas and propositional logic reasoning (modus ponens etc) is very convenient.
- Intermediate individual reasoning steps are not recorded/displayed.
  - Mostly unnecessary, sometimes desired (proof "debugging").
- Automated arithmetic reasoning is restricted to linear arithmetic.
  - $a(b+1) = ab + b$ cannot be proved.
  - Semi-decision procedures (computer algebra) would be helpful.

All in all, the integration of automated rule-based reasoning with interactive human assistance and semi-automatic decision procedures yields a usable tool for use in classroom and elsewhere.