TR 2011/001 ISSN 0874-338X

THedu'11
CTP Components for Educational Software
Proceedings

Pedro Quaresma
Department of Mathematics
University of Coimbra, Portugal
&
Ralph-Johan Back
Department of Information Technologies
Åbo Akademi University
Turku, Finland

Center for Informatics and Systems of the University of Coimbra

THedu'11
CTP Components for Educational Software

Pedro Quaresma
CISUC/Department of Mathematics
University of Coimbra
3001-454 Coimbra, Portugal
e-mail: pedro@mat.uc.pt

Ralph-Johan Back
Department of Information Technologies
Åbo Akademi University
Turku, Finland
e-mail: backrj@abo.fi

2011/06/13

Workshop Description

This workshop intends to gather the research communities for Computer Theorem proving (CTP), Automated Theorem Proving (ATP), Interactive Theorem Proving (ITP) as well as for Computer Algebra Systems (CAS) and Dynamic Geometry Systems (DGS).

The goal of this union is to combine and focus systems of these areas to enhance existing educational software as well as studying the design of the next generation of mechanised mathematics assistants. Elements for next-generation MMAs include:

- Declarative Languages for Problem Solution: education in applied sciences and in engineering is mainly concerned with problems, which are understood as operations on elementary objects to be transformed to an object representing a problem solution. Preconditions and postconditions of these operations can be used to describe the possible steps in the problem space; thus, ATP-systems can be used to check if an operation sequence given by the user does actually present a problem solution. Such "Problem Solution Languages" encompass declarative proof languages like Isabelle/Isar or Coq's Mathematical Proof Language, but also more specialized forms such as, for example, geometric problem solution languages, that express a proof argument in Euclidean Geometry or languages for graph theory.

- Consistent Mathematical Content Representation: libraries of existing ITP-Systems, in particular those following the LCF-prover paradigm, usually provide logically coherent and human readable knowledge. In the leading provers, mathematical knowledge is covered to an extent beyond most courses in applied sciences. However, the potential of this mechanised knowledge for education is clearly not yet recognised adequately: renewed pedagogy calls for enquiry-based learning from concrete to abstract — and the knowledge's logical coherence supports such learning: for instance, the formula 2π depends on the definition of reals and of multiplication; close to these definitions are the laws like commutativity etc. However, the complexity of the knowledge's traceable interrelations poses a challenge to usability design.

- User-Guidance in Stepwise Problem Solving: Such guidance is indispensable for independent learning, but costly to implement so far, because so many special cases need to be coded by hand. However, CTP technology makes automated generation of user-guidance reachable: declarative languages as mentioned above, novel programming languages combining computation and deduction, methods for automated construction with ruler and compass from specifications, etc — all these methods 'know how to solve a problem'; so, use the methods' knowledge to generate user-guidance mechanically, is an appealing challenge for ATP and ITP, and probably for compiler construction!

In principle, mathematical software can be conceived as models of mathematics: The challenge addressed by this workshop is to provide appealing models for MMAs which are interactive and which explain themselves such that interested students can independently learn by enquiry and experimentation.

The workshop received thirteen submissions, twelve of which were accepted and contained in these proceedings. The submissions are within the scope of the following points, which have been announced in the call of papers.

- CTP-based software tools for education in:
  - proving within math courses;
  - problem solving in applied sciences;
  - applied sciences with impressive examples;
  - specific domains like geometry, graph theory, etc.
- CTP technology combined with novel interfaces, drag and drop, etc.
- technologies to access ITP knowledge relevant for a certain step of problem solving;
- usability considerations on representing ITP knowledge related to a step:
  - make traces of (a bulk of) rewrites comprehensible for humans (grouping etc);
  - visualize relations between deductive knowledge and algorithmic knowledge.
- combination of deduction and computation:
  - environments (a la programming) — logical contexts;
  - check user input against a logical context by ATP;
  - resume computation after user input.
- formal problem specifications:
  - abstraction of pre-conditions and post-conditions to problem classes;
  - arrangement of related problem classes (trees, graphs);
  - management of specifications (hide from beginners, reveal on request);
- effectiveness of ATP in checking user input:
  - examples and counter-examples;
  - general techniques and limitations.
- formats for deductive content in proof documents, geometric constructions, etc;
- formal domain models for e-learning in mathematics and applications.

The above list for topics of interest will be revised during the workshop as well as the workshop description on the first page. Looking forward to a fruitful workshop at CADE!

The program committee of THedu'11

Contents

THedu'11 Program

Sunday, 2011-07-31

09:00-09:30 Opening

09:30-10:00 Multilinguality in MMAs

Session Chair: Walther Neuper

- Jordi Saludes and Sebastian Xambó. *The GF Mathematics Library.*

10:00-10:30 Coffee Break

10:30-12:30 ITPs & Formalizations

Session Chair: Laurent Théry

- Makarius Wenzel and Burkhart Wolff. *Isabelle/PIDE as Platform for Educational Tools.*
- Guillaume Allais. *Coq with power series.*
- Cezary Kaliszyk. *Formalized Computational Origami in Education.*
- Filip Marić, Predrag Janičić, Ivan Petrović and Danijela Petrović. *Formalization and Implementation of Algebraic Methods in Geometry.*

12:00-14:00 Lunch

14:00-15:30 Combination of Tools

Session Chair: Wolfgang Schreiner

- Francisco Botana and Miguel A. Abánades. *Automatic Deduction in Dynamic Geometry using Sage.*
- Vladimir Komendantsky. *Packed views of pre-structured data.*
- Vanda Santos and Pedro Quaresma. *WebGeometryLab.*

15:30-16:00 Coffee Break

16:00-18:00 Semantics of Programs & Methodological Questions

Session Chair: Pedro Quaresma

- Wolfgang Schreiner. *Computer-Assisted Program Reasoning Based on a Relational Semantics of Programs.*
- Ralph-Johan Back and Johannes Eriksson. *Correct-by-construction programming in the Socos environment.*
- Walther Neuper *User Guidance Generated from "Computation plus Deduction" – the Learner's Perspective.*
- Serge Autexier, Dominik Dietrich and Marvin Schiller. *Cognitive Tutoring in Mathematics based on Assertion Level Reasoning and Proof Strategies.*

Monday, 2011-08-01

18:30 Business Meeting

Coq with power series

Guillaume Allais
Junior Laboratory COQTAIL
Ens Lyon - France
guillaume.allais@ens-lyon.org

In the interactive theorem prover Coq[1], trigonometric functions are defined in the standard library. However they are not directly described as power series due to a lack of formalization of this concept. We present a strategy to describe the power series over $\mathbb{R}$ and then advocate for the broad use of abstract concepts in the standard library by showing the immediate benefits it brings.

Thanks to the power series framework we are able to redefine the usual functions (e.g. sin, cos, exp) in less than 80 lines, to get some of their properties for free (e.g. being in the C^∞ class) and to prove that they are solutions of particular differential equations just by studying sequences over $\mathbb{R}$.

Files: All the implementation mentioned in this paper are available for download via COQTAIL's svn repository[1].

1 Formalization

Power series are a rather intensional mathematical notion; therefore, its formalization in an interactive theorem prover is really close to textbooks' presentation. Except for one point: the definition of the convergence radius.

In an intuitionistic setting, our definition of the radius of convergence is much more informative than the standard one. This is however not harmful given that it has been proved classically equivalent to the standard one. This design choice allows us to get rid of the excluded-middle (henceforth EM) axiom in most proofs that traditionally use it. One could expect as a drawback that the lemmas with the existence of a radius of convergence as a conclusion not to be provable anymore. It is fortunately not the case[2].

1.1 The convergence radius

The convergence radius ρ of a power series whose coefficients are $(a_n)_{n\in\mathbb{N}}$ is usually defined as a lowest upper bound (in $\mathbb{R}\cup\{+\infty\}$):

$$\rho\left(\sum_{n\in\mathbb{N}} a_n x^n\right) = \sup\{r\in\mathbb{R} \mid \text{the sequence } |a_n r^n| \text{ is bounded}\}$$

As being bounded is not decidable, knowing that r is the convergence radius of $\sum_{n\in\mathbb{N}} a_n x^n$ is not sufficient to show without using EM that the sequence $(|a_n x^n|)_{n\in\mathbb{N}}$ is bounded for all x in $B_r(0)$. That is why we use a more verbose definition which describes exactly the same idea of being the lowest upper bound and is easier to use.

[1] see `http://sourceforge.net/projects/coqtail/develop` and in particular `src/Reals/Rpser.v`.
[2] At least for d'Alembert's ratio criterion which is one of the key lemmas that have this shape.

Submitted to:
THedu 2011

Definition 1 (Rpser_def) *We say that r is a weak convergence radius if it is a lower bound for the convergence radius (ie. r belongs to the disk of convergence).*

$$\texttt{Cv\_radius\_weak}\,(a_n, r) = |a_n r^n| \text{ is bounded}$$

From this definition, we can obtain the definition of the finite convergence radius[3].

Definition 2 (Rpser_def) *The convergence radius is finite and equal to r if r is both bigger than all the weak convergence radiuses and smaller or equal to all the reals that are too big to be weak convergence radiuses.*

$$\texttt{finite\_cv\_radius}\,(a_n, r) = \begin{array}{lll} & \forall r', \quad 0 \leq r' < r & \Rightarrow \texttt{Cv\_radius\_weak}\,(a_n, r') \\ \wedge & \forall r', \quad r < r' & \Rightarrow \neg\, \texttt{Cv\_radius\_weak}\,(a_n, r') \end{array}$$

The classical equivalence between this definition and the usual one is proved through the two following lemmas. Unsurprisingly, the first implication does not need the excluded middle axiom: our definition is stronger in an EM-free setting than the usual one.

Lemma 1 (Rpser_base_facts) $\texttt{finite\_cv\_radius}\,(a_n, r) \Rightarrow r = \sup\{x | \texttt{Cv\_radius\_weak}\,(a_n, x)\}$

Lemma 2 (Rpser_base_facts) $EM \wedge r = \sup\{x | \texttt{Cv\_radius\_weak}\,(a_n, x)\} \Rightarrow \texttt{finite\_cv\_radius}\,(a_n, r)$

The first important tool that we can get is d'Alembert's ratio criterion. It can be used to prove that a particular power series has a given convergence radius; this result is for example used to prove that exp has an infinite convergence radius from which we can easily deduce that so do cos and sin.

Lemma 3 (Rpser_cv_facts) *D'Alembert's ratio criterion states that, given a sequence* $(a_n)_{n\in\mathbb{N}}$ *which ultimately contains only nonzero elements:*

$$\lim_{n\to+\infty} \left|\frac{a_{n+1}}{a_n}\right| = \lambda \neq 0 \Rightarrow \texttt{finite\_cv\_radius}\,(a_n, \frac{1}{\lambda})$$

1.2 Sum of a power series

When we know what is the power series' convergence radius, we can start summing it on the appropriate domain. Our cherished tool to define the sum of a power series is obviously Abel's lemma which states that given a convergence radius, we can sum the power series inside the corresponding ball.

Lemma 4 (Rpser_radius_facts)

$$\texttt{Cv\_radius\_weak}\,(a_n, r) \Rightarrow \forall x \in B_r(0), \exists l, \sum_{n=0}^{+\infty} a_n x^n = l$$

Definition 3 (Rpser_sums) *From this lemma we can construct the functions (namely* `weaksum_r`*,* `sum_r` *and* `sum`*) that given either* $\texttt{Cv\_radius\_weak}\,(a_n, r)$*,* $\texttt{finite\_cv\_radius}\,(a_n, r)$ *or* $\texttt{infinite\_cv\_radius}\,(a_n)$ *output the piecewise-defined function:*

$$x \mapsto \begin{cases} \sum_{n=0}^{+\infty} a_n x^n & \text{if } x \text{ is inside the convergence disk} \\ 0 & \text{otherwise} \end{cases}$$

[3]The extension to the infinite case is straightforward: the convergence radius is infinite if and only if all the reals are weak convergence radiuses.

1.3 Derivative of a power series

The formalization of the derivative of a power series is done in two steps. First of all, we define the formal derivative of a power series and prove that this definition makes sense (e.g. that the sum exists):

Definition 4 (Rpser_derivative) *The formal derivative of the power series defined by (a_n) is the one defined by:*

$$\texttt{An\_deriv}\,(a_n) = (n+1) * a_{n+1}$$

Lemma 5 (Rpser_radius_facts) *And its convergence radius is exactly the same as the one of the original series:*

$$\texttt{finite\_cv\_radius}\,(a_n, r) \Leftrightarrow \texttt{finite\_cv\_radius}\,(\texttt{An\_deriv}\,(a_n), r)$$

Now we know that the formal derivative is summable but we still have to somehow explicit the relation that exists between these two power series. This is maybe the most complex part of the library; it uses a theorem on sequences of functions (see infra) plus the fact that the sequence of the partial sums of a power series converges normally (thus uniformly) inside the disk of convergence.

Theorem 1 (RFsequence_facts) *On the ball $B_r(c)$ if $(f_n)_{n\in\mathbb{N}}$ is a sequence of derivable functions and if the following limits exist:*

$$f_n \xrightarrow{n\to+\infty} f \quad \textit{and} \quad f_n' \xrightarrow[CVU]{n\to+\infty} g$$

then f is derivable and its derivative is g.

As a consequence, we can conclude that the power series are derivable and that their derivative is precisely the sum of the formal derivative. As the derivative of a power series is another power series, it is trivial to show that the function defined as the sum belongs to the C^∞ class[4]. A simple induction can even give us the explicit description of the n^{th} derivative of a sum:

Definition 5 (Rpser_def) *The formal k^{th} derivative of the power series defined by (a_n) is the one defined by:*

$$\texttt{An\_nth\_deriv}\,(a_n, k) = \frac{(n+k)!}{n!} a_{n+k}$$

Lemma 6 (Rpser_derivative_facts) *Explicit description of the k^{th} derivative of the power series defined by $(a_n)_{n\in\mathbb{N}}$:*

$$\left(\sum_{n=0}^{+\infty} a_n x^n\right)^{(k)} = \sum_{n=0}^{+\infty} \texttt{An\_nth\_deriv}\,(a_n, k) x^n$$

2 Applications

2.1 Defining usual functions

Thanks to all the formalization work, we can now define all the usual functions in a few lines. We begin by defining the exponential using d'Alembert's ratio test to prove that its convergence radius is infinite (which is rather easy).

Then we can show easily that cosine and sine also have an infinite convergence radius by using one of the basic lemmas:

[4] See the C_n_* files for results on classes of functions.

Lemma 7 (Rpser_base_facts) `Cv_radius_weak` *is compatible with the pointwise order on the sequences over reals:*

$$(|b_n| \leq_{pointwise} |a_n|) \Rightarrow \forall r, \texttt{Cv\_radius\_weak}(a_n, r) \Rightarrow \texttt{Cv\_radius\_weak}(b_n, r)$$

We have been able to define exp, sin and cos in less than 80 lines of Coq source code. It has to be compared to the hundreds of lines and the ad-hoc arguments (e.g. convergence of alternating series) that Coq's standard library dedicates to the definition of these exact same functions.

Not only defining these functions is way easier, but we get for free their derivability (and even their being in the C^∞ class), the exact shape of their n^{th} derivative and therefore we can easily prove the relations that exist between them.

2.2 Finding solutions of linear differential equations

On top of the power series formalization, we constructed a small library that describes linear differential equations of arbitrary shape by reflection. It is based on two components:

- A datatype `side_equa` that is a descriptor of linear differential equations. A differential equation is a pair (E_1, E_2) of side equations (usually written $E_1 :=: E_2$).
- A (set of) semantics that given an inhabitant of this datatype and an environment, outputs a proposition.

Definition 6 (Dequa_def) *The grammar of the* `side_equa` *datatype is (where c is a real constant):*

$$E ::= c \mid y_i^{(k)} \mid -E \mid E + E$$

Definition 7 (Dequa_def) *The semantics that translates differential equations into propositions on sequences over* $\mathbb{R}$ *is defined in two steps. First of all we define recursively the interpretation of the side of an equation:*

$$\begin{array}{lcl}
\texttt{interp_N} & : & \texttt{side_equa} \to \texttt{Rseq list} \to \texttt{Rseq} \\
\texttt{interp_N}(\quad c \quad, \rho) & = & \texttt{An_cst}(c) \\
\texttt{interp_N}(\quad y_i^{(k)} \quad, \rho) & = & \texttt{An_nth_deriv}(\rho(i), k) \\
\texttt{interp_N}(\quad -E \quad, \rho) & = & -\texttt{interp_N}(E, \rho) \\
\texttt{interp_N}(\quad E_1 + E_2 \quad, \rho) & = & \texttt{interp_N}(E_1, \rho) + \texttt{interp_N}(E_2, \rho)
\end{array}$$

and then we can construct the interpretation function:

$$[|E_1 :=: E_2|]_{\mathbb{N}} \rho = \forall n, \texttt{interp\_N}(E_1, \rho)(n) = \texttt{interp\_N}(E_2, \rho)(n)$$

We can define similarly the semantics $[|\_|]_{\mathbb{R}}$ that translates differential equations into propositions on power series. Having this generic representation of differential equations and this different semantics allows us to state general theorems about linear differential equations.

Lemma 8 (Dequa_facts) *Our main result states that given a context* ρ *of sequences of coefficients, we have (provided that the involved power series have an infinite radius of convergence):*

$$[|E_1 :=: E_2|]_{\mathbb{N}} \rho \Rightarrow [|E_1 :=: E_2|]_{\mathbb{R}} \rho$$

ie. we can prove that some functions (sums of power series) are solutions of a given differential equation by proving results on their coefficients.

By using this theorem, one can prove in less than 20 lines that the exponential is a solution of the equation $y^{(n+1)} = y^{(n)}$. Without much more work, one can also prove that cosine and sine are solutions of $y^{(2)} = -y$.

3 Extensions

3.1 Power series on other carriers

COQTAIL already has a basic library on the power series over the complex numbers[5] (definitions, differentiability, derivability). A future work could be to adapt the work done on the real numbers to the complex numbers. A better idea might be to formalize the power series on a more general structure (e.g. a ring equipped with a norm).

3.2 Improving the library on differential equations

Even if the work on the differential equations is already quite convenient, it is still rather limited: one can only derive variables and not composed expressions, there is currently no built-in minus function or multiplication by a constant and it is impossible to talk about the product of power series.

In the future, a description of the explicit shape of the product of two power series (the power series which is defined with the Cauchy product of the coefficients' sequences) could be a valuable addition.

References

[1] INRIA: *The Coq proof assistant*.

4 Acknowledgements

Thanks to all the COQTAIL project's members who helped proofreading the drafts. Special thanks to Sylvain Dailler and Marc Lasson for their useful comments and suggestions.

[5] See src/Complex/Cpser_*.v

Cognitive Tutoring in Mathematics based on Assertion Level Reasoning and Proof Strategies

Serge Autexier Dominik Dietrich Marvin Schiller
German Research Centre for Artificial Intelligence (DFKI) Brunel University
autexier@dfki.de dominik.dietrich@dfki.de Marvin.Schiller@brunel.ac.uk

To know how to do proofs is a skill that is essential for every mathematician and scientist. Hence, learning how to do proofs is a major part in the education of students of mathematics and modern science. Intelligent tutoring systems (ITS) are an attractive vehicle to make high-quality teaching and training environments available for a wide public. There are two main approaches: Model tracing tutors (MTTs), which are process-centric and try to fathom the process a student arrived at a solution, and constraint based tutors (CBTs), which are product-centric and are based on the idea that diagnostic information is not in the sequence of actions leading to the problem state, but solely in the problem state itself (see [11, 10] for a comparison). According to [10], the main advantage of a CBT is that its design requires less time and effort (as it does not need to solve the problem itself), while a MTT offers more specific advice giving capabilities (as the solution building process is explicit).

There exist already strong tools for teaching mathematical computations (such as [9]) or pure logic (such as the CMU proof tutor [14] or Proofweb [8]), however there are only a few attempts that support teaching of mathematics at a more abstract level, comparable to the type of mathematics taught in schools or in the first year at university. The main reasons we identified for this are:

Automated Proof Search Limitations: Domain reasoning quickly becomes too difficult for existing reasoners. However, MTT require a domain reasoner to determine how a student arrived at a certain step.

Solution Space Size: A theorem may have different kinds of proofs, each of which can be formulated in slightly different variants. Thus, the ITS technique of preauthoring solutions (including problem- and situation-specific hints) is unsuitable – if not unfeasible – for teaching mathematical theorem proving.

Proof Search vs. Proof Presentation: Finding a proof for a conjecture is different from presenting the found proof to the user: for the final presentation, backward steps are often converted to forward steps, several steps are combined to obtain a shorter proof, and the names of the employed proof techniques are omitted (see [15] p. 13-15 for a discussion). Therefore, existing proofs can only be used as target for tutoring to a very limited extent.

Proof Step Granularity: Human tutors may reject a proof step even though it is logically correct as it may lack other desirable properties, for instance if it is of inappropriate step size or irrelevant with respect to a specific proof strategy to be taught. However, classical reasoners usually provide large proof objects based on some particular logical calculus, such as resolution, which make it difficult to judge the general appropriateness of a proof step proposed by a learner in a tutorial context or to synthesize a hint if the student requests help.

Human-style vs. Machine-style Proofs: Proof techniques used in standard automated theorem proving (ATP) are different from techniques used by humans to solve problems – it is unreasonable to expect the solutions generated by standard ATP systems to reflect these techniques. Therefore, inspecting the proof object makes it difficult to analyze the solution with respect to other properties than correctness. Moreover, the complexity of the generated proof objects make it difficult to extract suitable hints at a strategic level.

Submitted to:
THEdu 2011

Irrelevant/Inexpedient Proof States: Classifying (incorrect) problem states by constraints as done in CBT seems to be difficult in the domain of proof tutoring.

Therefore, we advocate a dynamic approach to support the computer assisted teaching of mathematical proofs in the spirit of MTT. In previous work, we have shown that the so-called *assertion level* (see [7]) provides a suitable basis to verify underspecified and incomplete human-level proof steps in the domain of set theory based on a proof reconstruction approach (see [5]). In contrast to machine oriented calculi, each proof step at the assertion level corresponds to the application of a theorem, definition, or axiom. Our assertion level prover systematically converts declarative knowledge (formulas) to procedural knowledge (inferences), a process that usually needs to be done manually in the context of MTT. The contribution of this paper is to give a concise summary of our work in the context of computer theorem proving, highlighting the techniques from the theorem proving community that are used. These are: The different processing models of the prover, which actively processes declarative proof commands entered by a student in a tutoring setting, and lazily processes proof commands in a pure verification setting (Section 1); specification of declarative domain knowledge in theory languages, the use of a declarative proof language to encode student dialogue turns, specification of strategic problem solving knowledge as well as verification techniques in proof strategies/tactics (see [6, 2] for an overview) in Section 2; and representation of proof reconstructions in a hierarchical proof data structure to analyze the student's input and to offer hints at different levels of detail (see [1] for details) in Section 3.

1 Proof Command Processing Models

In a tutoring context, the student stepwise enters proof steps to solve a given tutorial exercise in the domain of set theory. Proof steps are formulated as proof commands in a declarative proof language. However, the processing of a proof command by the proof assistant within the setting of tutorial dialogues differs from the processing in a pure verification setting with respect to the following points:

- In a pure verification setting, it is sufficient to find some verification for a proof command. The verification itself is usually not of interest and needs not to be further processed. In contrast, in a tutorial setting we need to consider several, if not all, possible verifications of the given proof command (and relate them to the knowledge of the student) and need to further analyze the verification to avoid the student to rely on the power of the underlying theorem prover to solve the exercise.
- In a pure verification setting, we can assume the user to be an expert in the problem domain as well as in the field of formal reasoning. This has several implications on the processing model: (i) inputs can be expected to be correct and just need to be checked, (ii) proof commands can lazily be verified until a (sub)proof is completed, (iii) justification hints are given that indicate how to verify a given proof command, (iv) feedback is limited to "checkable" or "not checkable". In contrast, in a tutorial setting, we must assume the user to be neither a domain expert nor an expert in formal reasoning. The underlying mechanisms need to be hidden from the user, direct and comprehensive feedback has to be provided at each step. Therefore, it is for example a requirement to anticipate why an assumption is made, in contrast to a lazy checking once the conclusion has been obtained.
- In a pure verification setting, we can assume the user to indicate when the proof of a subgoal is finished (as usually done by so-called *proof step markers* in the proof language). However, in the tutorial setting this information is implicit. Similarly, we must be able to perform backward steps where some of the new proof obligations have not yet been shown.

```
strategy work-backward
  repeat
  use select * from definitions as
backward

strategy close-by-logic
  repeat first deepaxiom, or-l
```

```
strategy work-forward
  repeat
  use select * from definitions as forward

strategy close-by-definition
  try work-backward then try work-forward
then close-by-definition
```

Figure 1: Formalization of a simple proof strategy

2 Representation of Proof Strategies

A *proof strategy* represents some mathematical technique that happens to be typical for a given problem. For example, there are strategies which perform proof by induction, proof by contradiction, solve equations, or unfold definitions. To achieve a goal, a strategy performs a heuristically guided search using a dynamic set of assertions, as well as other strategies. Proof strategies are encoded in a strategy language (see [6, 2] for an overview) and generate hierarchical *strategic proof plans* (see [1] for details), where each justification in a plan corresponds to a strategy or an assertion application. The hierarchies arise from invocations of strategies or assertion applications from inside strategies. Intuitively, a hierarchical proof plan at a high level sketches how the overall problem was structured into subproblems at a meta level. At the lowest level, a concrete proof with concrete assertion steps is given.

A simple proof strategy that is proposed in [15] is the "Forward-Backward Method", which consists of first applying definitions to the conclusion of a goal to simplify it, and then using forward reasoning on the assumptions to derive new facts and finally close the goal. We have formalized this method within our strategy language within the strategy "Close-by-Definition" which relies on three sub-strategies, "Work-Forward", "Work-Backward" and "Close-by-Logic". "Work-Forward" works on the assumptions of the current task and mainly expands definitions. "Work-Backward" tries to simplify the current goal by applying definitions. "Close-by-Logic" applies logical reasoning, such as performing case splits, to close the task it was applied to. Figure 1 shows the formalization of the methods.

3 Adaptive Generation of Hints

The basic idea to dynamically generate context sensitive hints for a specific proof task is simple: Complete the proof for the current proof task using a proof strategy (such as close by definition) and analyze the solution to extract and generate a suitable hint. Thereby, make use of the proof hierarchies to increase the detailedness of hints on demand. More precisely, the generation of a hint works as follows: (i) selecting a certain level of hierarchy in the reconstruction, (ii) selecting a sequence of steps starting from the task with which the completion of the proof was invoked, (iii) extracting information from this sequence and converting it to a concrete hint. A given hint can be refined by either switching to a more detailed level of hierarchy, increasing the length from which the hint was generated, or by increasing the information which was extracted from the selected sequence. Let us stress here that this simple approach is only made possible by our abstract proof presentation at the assertion level and would be much more complicated if not impossible within natural deduction or in a resolution calculus.

To illustrate how such a strategy can be used to generate a context-sensitive hint, consider the exercise $(R \circ S)^{-1} = S^{-1} \circ R^{-1}$. Suppose that the student starts the proof by applying the definition of set equality,

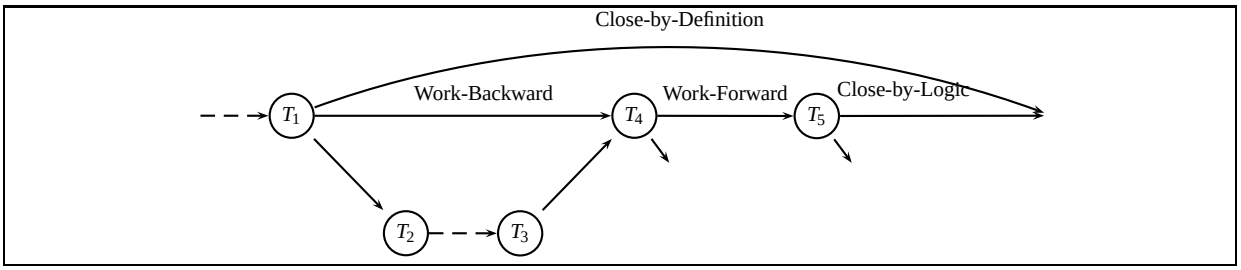


Figure 2: Hierarchical proof plan completing the proof of task T_1

yielding two subtasks

$$T_1 : \vdash (R \circ S)^{-1} \subset S^{-1} \circ R^{-1} \tag{1}$$

$$T_1' : \vdash S^{-1} \circ R^{-1} \subset (R \circ S)^{-1} \tag{2}$$

and requests a hint for the task T_1. A possible completion of the proof, encoded in the strategy "Close-by-Definition", consists of expanding all definitions and then using logical reasoning to complete the proof. The resulting hierarchical proof object is shown schematically in Figure 2. The task T_1 has three outgoing edges, the topmost two corresponding to a strategy application and the lower-most one corresponding to an assertion application, respectively. Internally the edges are ordered with respect to their granularity, according to the hierarchy of nested strategy applications that generated them. In the example the most abstract outgoing edge of T_1 is the edge labelled with "Close-by-Definition", followed by the edge labelled with "Work-Backward", both representing strategy applications. The edge with the most fine-grained granularity is the edge labelled with "Def $\subset$" and represents an inference application.

By selecting the edges "Work-Backward", "Work-Forward", and "Close-by-Logic", we obtain a flat graph connecting the nodes T_1, T_4, and T_5. A more detailed proof-view can be obtained by selecting the edge "Def $\subset$" instead of "Work-Backward". In this case the previous single step leading from T_1 to T_4 is replaced by the subgraph traversing T_2 and T_3. Each selection can be used to generate several hints. Suppose for example that we select the lowest level of granularity, and the first proof state to extract a hint. This already allows the generation of three hints, such as

- "Try to apply Def $\subset$"
- "Try to apply Def $\subset$ on $(R \cup S) \circ T \subset (T^{-1} \circ S^{-1})^{-1} \cup (T^{-1} \circ R^{-1})^{-1}$"
- "By the application of Def $\subset$ we obtain the new goal $(x,y) \in (R \cup S) \circ T \Rightarrow (x,y) \in (T^{-1} \circ S^{-1})^{-1} \cup (T^{-1} \circ R^{-1})^{-1}$"

Selecting a more abstract level would result in hints like "Try to work backward from the goal", or "Try to apply definitions on the goal and assumptions". Within our approach, the analysis functions to extract hints and their verbalization are hard-coded within the programming language.

4 Evaluation

The presented techniques have been successfully evaluated in experiments with computer-based tutoring of proofs in naive set theory [3]. In [12], we have shown that reconstructions at the assertion level can be used for a proof step analysis that goes beyond the correctness of proof steps, such as the analysis of proof granularity, i.e. the question whether a given proof step is of appropriate size. Moreover, in [13]

we have advocated that assertion-level proof steps provide a good level of abstraction to approximate and judge the proof steps done by students in the naive set theory domain.

References

[1] Serge Autexier, Christoph Benzmüller, Dominik Dietrich, Andreas Meier & Claus-Peter Wirth (2006): *A Generic Modular Data Structure for Proof Attempts Alternating on Ideas and Granularity*. In: *Proc. of MKM*, Springer, Bremen, pp. 126–142.

[2] Serge Autexier & Dominik Dietrich (2010): *A Tactic Language for Declarative Proofs*. In Matt Kaufmann & Lawrence C. Paulson, editors: *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, *Lecture Notes in Computer Science* 6172, Springer, pp. 99–114. Available at `http://dx.doi.org/10.1007/978-3-642-14052-5_9`.

[3] Christoph Benzmüller, Helmut Horacek, Henri Lesourd, Ivana Kruijff-Korbayova, Marvin Schiller & Magdalena Wolska (2006): *A corpus of tutorial dialogs on theorem proving; the influence of the presentation of the study-material*. In: *Proceedings of International Conference on Language Resources and Evaluation (LREC 2006)*, ELDA, Genova, Italy, pp. 1766–1769.

[4] Peter Brusilovsky, Albert T. Corbett & Fiorella de Rosis, editors (2003): *User Modeling 2003, 9th International Conference, UM 2003, Johnstown, PA, USA, June 22-26, 2003, Proceedings*. *Lecture Notes in Computer Science* 2702, Springer.

[5] Dominik Dietrich & Mark Buckley (2008): *Verification of Human-level Proof Steps in Mathematics Education*. *Teaching Mathematics and Computer Science* 6(2), pp. 345–362.

[6] Dominik Dietrich & Ewaryst Schulz (2009): *Integrating Structured Queries into a Tactic Language*. *JAL - Special issue on Programming Languages and Mechanized Mathematics Systems* .

[7] Xiaorong Huang (1994): *Reconstructing Proofs at the Assertion Level*. In Alan Bundy, editor: *Proc. 12th CADE*, Springer-Verlag, pp. 738–752. Available at `citeseer.ist.psu.edu/huang94reconstructing.html`.

[8] Cezary Kaliszyk, Freek Wiedijk, Maxim Hendriks & Femke van Raamsdonk (2007): *Teaching logic using a state-of-the-art proof assistant*. In H. Geuvers & P. Courtieu, editors: *Proc. of the International Workshop on Proof Assistants and Types in Education*, pp. 33–48.

[9] Erica Melis, Eric Andrès, Jochen Büdenberger, Adrian Frischauf, George Goguadze, Paul Libbrecht, Martin Pollet & Carsten Ullrich (2001): *ActiveMath: A Generic and Adaptive Web-Based Learning Environment*. *Artifical Intelligence in Education* 12(4).

[10] Antonija Mitrovic, Kenneth R. Koedinger & Brent Martin (2003): *A Comparative Analysis of Cognitive Tutoring and Constraint-Based Modeling*. In Brusilovsky et al. [4], pp. 313–322. Available at `http://link.springer.de/link/service/series/0558/bibs/2702/27020313.htm`.

[11] Antonija Mitrovic & Stellan Ohlsson (2006): *A Critique of Kodaganallur, Weitz and Rosenthal, "A Comparison of Model-Tracing and Constraint-Based Intelligent Tutoring Paradigms"*. *I. J. Artificial Intelligence in Education* 16(3), pp. 277–289. Available at `http://iospress.metapress.com/content/1rlwf2k7qdr1ag4l/`.

[12] Marvin Schiller (2010): *Granularity Analysis for Tutoring Mathematical Proofs*. Ph.D. thesis, Saarland University.

[13] Marvin Schiller & Christoph Benzmüller (2010): *Human-Oriented Proof Techniques are Relevant for Proof Tutoring*. Extended Abstract for MIPS 2010 workshop (affiliated with CICM 2010). CNAM, Paris, France.

[14] Wilfried Sieg & Richard Scheines (1994): *Computer Environments for Proof Construction*. *Interactive Learning Environments* 4(2), pp. 159–169.

[15] Daniel Solow (2005): *How to read and do proofs*. John Wiley and Sons.

Correct-by-construction programming in the Socos environment

Ralph-Johan Back Johannes Eriksson
Department of Information Technologies
Åbo Akademi University
Turku, Finland
backrj@abo.fi joheriks@abo.fi

Socos is an environment to support invariant-based programming, a correct-by-construction programming methodology in which the invariants are written before the code. Socos consists of a diagram editor connected to a theorem prover; it automatically generates and tries to prove them using the PVS theorem prover and the Yices SMT solver. We give an overview of the system and illustrate its use through two examples.

1 Introduction

Invariant-based programming (IBP) is a practically oriented approach to formal verification of imperative programs [3]. IBP is a *correct-by-construction* method: the correctness proofs are developed hand-in-hand with the program, rather than in a separate a posteriori verification phase. Additionally, the internal loop invariants are written *before* the code. After the invariants have been established, the code is added in small increments, while simultaneously verified to preserve the invariants. The main benefit of this approach is that the correctness proofs become closely connected to the code, which seems to make the program verification workflow easier to manage and learn [5, 3].

Any non-trivial program generates a significant number of lemmas to be proved, most of which are mathematically shallow. We have built a programming environment called Socos, which applies existing state-of-the-art automatic theorem proving (ATP) tools and satisfiability modulo theories (SMT) solvers to automatically discharge as many of the lemmas as possible. User interaction occurs through a graphical diagram editor, which supports both constructing the program and checking its correctness; this front-end is implemented as a plug-in for Eclipse [1]. Socos uses the theorem prover PVS [9] and the SMT solver Yices [6] as back-ends.

Our system allows use of the full higher-order logic of PVS in specifications and invariants. Hence, we cannot expect all conditions to be proved automatically. Remaining conditions can be proved interactively in the PVS proof assistant, or alternatively, automatic proof can be attained by introducing abstractions which are more suitable for automatic reasoning than the actual definitions. Such abstractions can be added to the system through *background theories*, and if defined in a way that the ATP or SMT solver can make use of them, may significantly increase the level of proof automation.

2 Invariant-based programming in the Socos environment

Socos supports the specification, implementation and checking of invariant-based programs. The programmer constructs and edits diagrams in a graphical environment using drawing program like gestures. By the click of a button, Socos generates verification conditions from the diagram and sends them to the

Submitted to:

automatic prover for simplification. Socos gives immediate feedback on the result of the verification and shows all unproved conditions to the programmer.

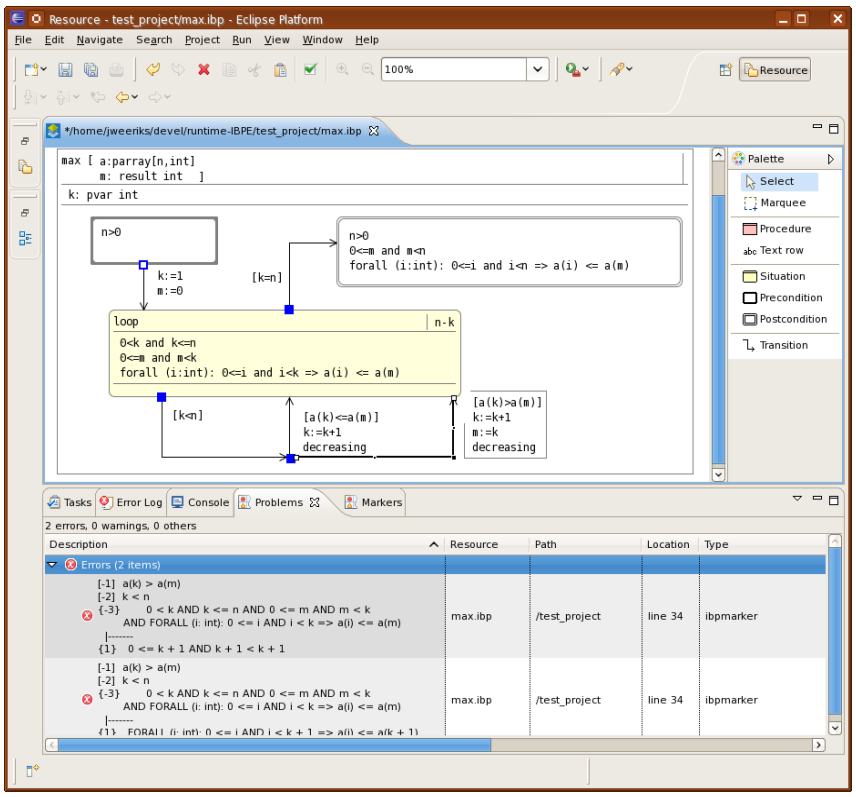


Figure 1: Invariant diagram (with errors) being checked in Socos

Figure 1 illustrates a Socos session in which a simple iterative program—linear search for the index of the maximal element in an integer array—is being checked. A procedure (`max`) is implemented as a statechart like graph—an *invariant diagram*—connecting an *initial situation* (the leftmost box) to a *final situation* (the rightmost box) through an intermediate situation (`loop`). A situation represents a named *state predicate*; it stands for all program states that satisfy the constraints listed inside the situation. Constraints are given in the PVS language [8]. A situations may be nested inside another situation, in which case it inherits all constraints of the enclosing situations. The initial, final and intermediate situations correspond respectively to the precondition, postcondition and loop invariants of the program.

Transitions—labeled arrows connecting to the edges of situations—constitute the actual program code. A transition can be labeled with a guard, an assignment statement, and a procedure call statement (the syntax of program statements is described in [7]). A transition is *consistent* if and only if it establishes all constraints in the target situation, assuming the constraints of the source situation holds together with the guards and assignments on the arrow. This *verification condition* is automatically calculated by Socos when the program is checked (based on weakest preconditions, see [7] for a description of the verification semantics), and translated into a PVS theorem and a proof script. The PVS proof checker executes the proof script, and all constraints not discharged are reported back to the user through the Eclipse "Problems view." Any PVS strategy can be used to attempt to discharge the lemma; the default is to invoke the Yices SMT solver (PVS prover command:`(yices)`).

The highlighted loop transition in Figure 1 has two unproved conditions, corresponding to the second and third constraint of the `loop` situation. The terms above the turnstile in a condition are the assumptions, while the term below the turnstile is the unproved constraint (with the updated variables substituted for their new values). It is easy to see that the two conditions in the above example are unprovable. This is due to an error in the second loop transition: the assignments to `k` and `m` occur in the wrong order, resulting in the given loop invariant not being preserved by the transition. If the order of the assignments is reversed, the program is proved automatically and no errors are reported.

In addition to consistency, Socos checks *liveness* and *termination*. Liveness ensures that the program does not terminate in an intermediate situation, i.e., at least one transition is always enabled in each non-final situation. A program is terminating if it can be proved that there are no infinite loops. Termination verification requires specifying a variant that is decreased on every cycle back to the recurring situation; the variant is written in the upper right hand corner of the situation and the decreasing transitions are indicated with the `decreasing` keyword.

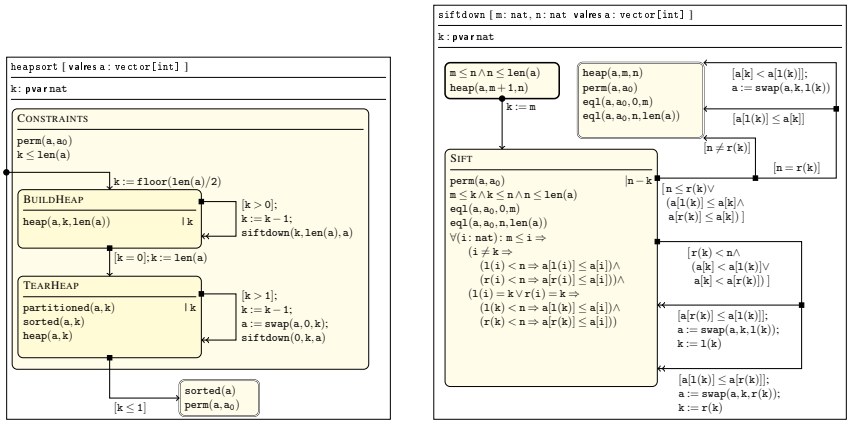
Diagrams can be built and verified incrementally (transition by transition). I.e., all transitions do not have to be in place when the program is checked. Consistency is the main property that should always hold for all transitions that have been added to the diagram. Verification of liveness and termination can be postponed until after the complete program has been verified to be consistent.

3 Background theories

An SMT solver is limited to the set of (decidable) background theories that it supports; for example, SMT solvers in general do not handle non-linear arithmetic or quantifiers. ATP strategies (such as `grind` in PVS) are better at dealing with quantifiers but do not always work. However, proof automation can in practice often be extended into new domains by identifying a few central theorems on which the correctness of a program depends, and sending these axioms to the decision procedure. We have built a PVS tactic that allows the default verification strategy to be extended with user-defined background theories developed in PVS [7].

Figure 2 shows a more complicated program: an implementation of heapsort consisting of the two procedures `heapsort` and `siftdown`. This program has 81 verification conditions to check, several of which involve the notion of permutation, a second order property which cannot be defined in an SMT solver. However, with the help of the background theory lemmas listed below the diagrams in Figure 2, our tactic automatically discharges 80 out of the 81 conditions associated with the procedures (we proved the final condition in PVS). These lemmas state that two permutations of an array have identical size, that permutation is an equivalence relation closed under pairwise swapping, and that the top element of a max-heap is maximal. The remaining lemma is the most difficult one to prove; it results from the call to `siftdown` in the second loop, requiring us to show that `partitioned` is maintained by `siftdown`.

Good background theories are challenging to develop. Not only must the included definitions be succinct and correct, they must also provide a vocabulary for writing clean and readable specifications and programs. Furthermore, the definitions and theorems must be expressed in a way that makes them usable by PVS. For a new domain we spend about 50% developing the background theories, while the other 50% is spent building the program (invariants and transitions). However, the time vested in developing background theories is amortized over a large number of programs in the same domain.



perm_len : **lemma** perm(a, b) ⇒ len(a) = len(b)
perm_ref : **lemma** perm(a, a)
perm_sym : **lemma** perm(a, b) ⇒ perm(b, a)
perm_trs : **lemma** perm(a, b) ∧ perm(b, c) ⇒ perm(a, c)
swap_perm : **lemma** ∀(i, j : index(a)) : perm(a, swap(a, i, j))
heap_max : **lemma** ∀(k : nat) : heap(a, 0, k) ⇒ (∀(i : nat) : 0 ≤ i ∧ i < k ⇒ a[i] ≤ a[0])

Figure 2: Socos implementation of heapsort and lemmas excerpted from the background theory

4 System description

Figure 3 shows the software architecture of Socos and its dependencies on external components. Dotted components represent work in progress. The back-end component (IBP-VC) reads textual invariant-based programs and is an interface to verification condition generation (VCG) and compilation (COMP). The former generates PVS theories containing all conditions for the checked program and a proof script that attempts the default strategy on each condition. It also interacts with the PVS prover and generates a report of proved and yet to be proved conditions. A compiler (COMP) from invariant diagrams into executable code is currently under construction.

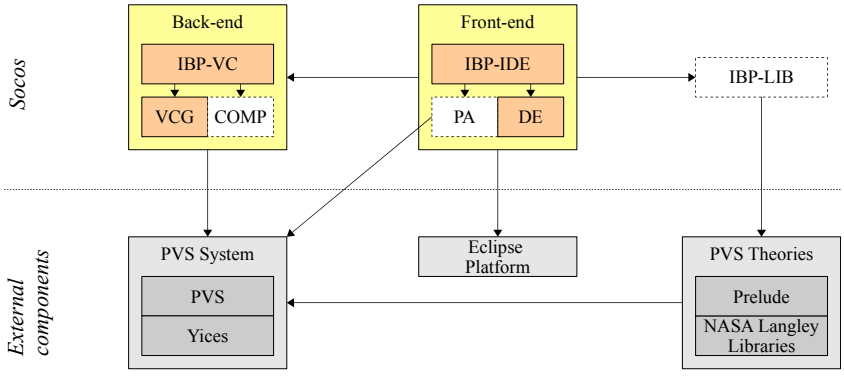


Figure 3: Architecture of Socos

The graphical front-end, IBP-IDE, is implemented as an Eclipse plug-in. We are also working on a proof assistant (PA), in which correctness proofs can be written as structured derivations [4] and checked

by PVS. The IBP-LIB is intended to be a library of background theories and strategies. It currently consists of just a few basic theories for arrays and vectors, but we plan on extending it based on a large number of case studies. These theories will be built on the PVS prelude and the comprehensive NASA Langley collection of theories [2].

Socos is available at `http://www.imped.fi/socos`.

5 Conclusion

Socos is an graphical programming environment built specifically to support IBP. It translates invariant diagrams into PVS correctness conditions, which are checked using the SMT solver Yices. It provides immediate feedback when checking fails, and pinpoints the unproved conditions. Domain-specific PVS background theories can simplify specification and improve proof automation.

IBP is a new programming methodology that has not been applied in large case studies yet. Good tool support is crucial for the methodology to scale up to programs of realistic size and complexity. So far, we have gathered a lot of experience from using IBP and Socos in teaching program verification at Åbo Akademi University, both at the basic level (second year undergraduate students) and advanced level (fourth year and graduate students). Feedback received from students indicates that they find the method easy to learn, and that it combines theorem proving and programming in an intuitive manner.

References

[1] Eclipse integrated development environment. `http://www.eclipse.org`.

[2] NASA Langley PVS Libs. `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html`.

[3] R.-J. Back. Invariant based programming: Basic approach and teaching experiences. *Formal Aspects of Computing*, 21 (3):227–244, 2009.

[4] R.-J. Back. Structured derivations: a unified proof style for teaching mathematics. *Formal Aspects of Computing*, 2009. Online at `http://www.springerlink.com/index/a75tmu1110kku422.pdf`.

[5] R.-J. Back, J. Eriksson, and L. Mannila. Teaching the construction of correct programs using invariant based programming. In *Proc. of 3rd South-East European Workshop on Formal Methods (SEEFM07)*. South-East European Research Centre (SEERC), 2007.

[6] B. Dutertre and L. de Moura. The Yices SMT solver. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, Aug. 2006. Available at `http://yices.csl.sri.com/tool-paper.pdf`.

[7] J. Eriksson and R.-J. Back. Applying PVS background theories and proof strategies in invariant based programming. In *Proc. of the 12th International Conference on Formal Engineering Methods (ICFEM'2010)*, Shanghai, China, Nov. 2010.

[8] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference, Version 2.4*. Computer Science Laboratory, SRI International, Menlo Park, CA, Nov. 2001.

[9] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In *Proc. of the Eighth International Conference on Computer Aided Verification (CAV'96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer, 1996.

Automatic Deduction in Dynamic Geometry using Sage

Francisco Botana
Departamento de Matemática Aplicada I, Universidad de Vigo, Campus A Xunqueira, 36005 Pontevedra, Spain
fbotana@uvigo.es

Miguel A. Abánades
CES Felipe II, Universidad Complutense de Madrid, 28300 Aranjuez, Spain
abanades@ajz.ucm.es

We present a symbolic tool that provides robust algebraic methods to handle automatic deduction tasks for a dynamic geometry construction. The prototype has been developed as two different worksheets for the open source computer algebra system Sage, corresponding to two different ways of coding a geometric construction. In one worksheet, diagrams constructed with the open source dynamic geometry system GeoGebra are accepted. In this worksheet, Groebner bases are used to either compute the equation of a geometric locus in the case of a locus construction or to determine the truth of a general geometric statement included in the GeoGebra construction as a boolean variable. In the second worksheet, locus constructions coded using the common file format for dynamic geometry developed by the Intergeo project are accepted for computation. The prototype and several examples are provided for testing.

1 Introduction

The name of Dynamic Geometry Systems (DGS) is given to the family of computer applications that allow exact on-screen drawing of (generally) planar geometric diagrams and, their main characteristic, the manipulation of these diagrams by mouse dragging certain elements making all other elements to automatically self adjust to the changes. This is also known as Interactive geometry.

Since the appearance of the French *Cabri* and the American *The Geometers Sketchpad* in the late 80s, many others have been created with slightly different functionalities (C.a.R. Euklides, Dr. Genius, Dr. Geo, Gambol, Geometrix, Geonext,...). Special mention deserve *Cinderella* [17] for its use of complex numbers methods as basis for its computations and *GeoGebra* [9], whose open source model and effective community development has resulted in a spectacular world wide distribution.

From the beginning, DGS have been the paradigm of new technologies applied to Math education. Being able to produce a great number of examples of a configuration has many times been taken as a substitute for a formal proof in what has come to be known as a *visual proof*. Questions have been raised on the influence of this use of DGS on the development of the concept of proof in school curricula [10]. This is a symptom of the incompleteness of general DGS relative to further manipulation of configurations. Although most DGS considered come equipped with some property checker, their numeric nature does not really provide a sound substitute for a formal proof.

To compensate the computational limitations of DGS, two main approaches have been taken to add symbolic capabilities to DGS. Some systems incorporate their own code to perform symbolic computations (e.g. [8]), while other systems, including several by the authors, choose to reuse existing Computer Algebra Systems (CAS) (e.g. [6, 3, 18, 20]). The prototype presented here offers a solution in this latter direction but with a relevant change: it is strictly based on open source tools.

Submitted to:
THedu 2011

Open source development, with its philosophy resembling a bazaar of different agendas and approaches, was thought at the beginning to be only good for small applications. Its success, clearly exemplified by the magnitude of the operating system GNU/Linux, came as a surprise to even the most optimistic programmers, who thought that big applications would always need a reverent cathedral-building approach. Nowadays, open source applications have impacted all computing areas and are no longer considered marginal.

If we take open as a synonym for accessible, we can not find fields where this concept is more relevant than Education, where universal access should be the leading principle, and Mathematics, where public scrutiny is at its very foundations. In consequence, the open source nature of the Math educative prototype that we present here, is not only a characteristic but a statement.

A few words on the main building blocks of our system, namely GeoGebra, Sage and Intergeo, are provided in the following sections. Details on how to use and test the system are provided in section 2.

1.1 GeoGebra

GeoGebra is an open source DGS with algebraic capabilities, establishing a direct relationship between the objects in the different windows: graphics, algebra, and spreadsheet. While its technical characteristics are of first order, what really made it our DGS of choice in this project was its impressive community integration that makes GeoGebra a *de facto* standard in the field.

GeoGebra was created in 2001 by Markus Hohenwarter as part of his Masters in Mathematics Education at the University of Salzburg (Austria). What was supposed to be a smaller tool, almost of personal use, won the 2002 Academy Award European software (EASA) in the category of Mathematics. Since then, the tool, using the word of mouth and Internet, spread rapidly throughout the world, and has become a collaborative project with impressive figures: users in 190 countries (The UN has 192 member states!), versions in 45 languages and half a million visitors to its web site every month.

It is worth mentioning the ongoing creation of a network of International GeoGebra Institutes (IGI) that serve as a platform from which teachers and researchers from around the world work together to promote the teaching of Mathematics. Currently there are 58 official IGIs in six continents (38 only in Europe, see [11] for a complete list).

1.2 Sage

Sage is an open source CAS designed to be a viable multiplatform free open source alternative to proprietary and expensive systems such as Mathematica, Maple or Matlab. The integration of multiple tools, the possibility of remote access via the Internet and the emphasis for decency and freedom make their most notable features.

Built out of nearly 100 open-source packages (including Singular, Axiom, Maxima,...), Sage features a unified interface that takes the form of a notebook in a web browser or the command line. Using the notebook, Sage connects either locally to your own Sage installation or to a Sage server on the network. Inside the Sage notebook you can create embedded graphics, beautifully typeset mathematical expressions, add and delete input, and share your work.

Sage was created in 2004 by William Stein [19], professor at the University of Washington, after several brushes over some accessibility issues with the developers of Magma, a highly specialized commercial CAS in whose development he had collaborated.

For its power and versatility we foresee Sage as the de facto standard for teaching mathematics with computers in secondary and university levels.

1.3 Intergeo

The *Intergeo* (i2g) file format is an XML-based specification designed to describe any construction created with a DGS. It is one of the main results of the Intergeo project, an eContentplus European project dedicated to the sharing of interactive geometry constructions across boundaries. For more information about the project, we refer to its website [12] and the documentation available there, as well as [15, 16].

An i2g file takes the form of a compress file package. In particular, intergeo.xml, provides a textual description of the construction in two parts, the elements part describing the (static) initial configuration and the constraints part where the geometric relationships are expressed. A detailed specification of the file format can be found at its webpage [7], where several examples are provided.

Among the list of elements covered by the format, we find several definitions of locus, used in our prototype. However no analogue of the GeoGebra boolean statement is considered (yet) by the i2g format. This is the reason why the Sage worksheet that computes the equation of geometric loci especified with the i2g format does not accept true/false queries.

2 Prototype Description

The prototype consists of two different Sage worksheets.

In ProofLocus4ggb.sws two different tasks are performed over GeoGebra constructions. Automatic deduction techniques based on Groebner bases (using Singular) are used to either compute the equation of a geometric locus in the case of a locus construction or to determine the truth of a general geometric statement included in the GeoGebra construction as a boolean variable. The worksheet includes a GeoGebra applet that allows the direct construction of a diagram or the upload of a local previously designed GeoGebra construction.

In Locus4i2g.sws, given a locus construction coded using the i2g file format, its equation and graph are provided.

The algorithm used to obtain the locus equation is due to the first author [4]. It has also been used by JSXGraph [13] and it is currently being used by GeoGebra to develope a similar function (see [5]).

This system presented here is a significantly upgraded version of the system to be presented by the first author in CICM [2]. Besides combining in a single worksheet both worksheets of that system, it adds the major advance of accepting i2g files. Its main technical difference, and the one that makes the user experience completely automatic, is the intensive use of JavaScript to allow the direct communication between Sage and the GeoGebra applet. This has made possible to circumvent the question/answer nature of Sage to generate what amounts to a one-click add-on for GeoGebra.

The system is presented in a prototype state and currently accepts a limited (easy to expand) set of construction primitives, namely free points, Midpoint(point-point), Point(on Circle and on Line), Segment(point-point), Line(point-point, point-line - meaning a parallel), OrthogonalLine, Circle(center-radius, center-point, center-radiusAsSegment), Intersect(object-object), Locus and Relation between Two Objects (parallelism, perpendicularity).

Both worksheets are available for testing online in the Sage server maintained by the authors (`https://193.146.36.205:9011/`, user: thedu11, password: test). Of course, the worksheets can also be downloaded for local use.

3 Examples

We illustrate the use of the prototype with two examples. The proof of a basic theorem in the form of a GeoGebra construction with a boolean variable and the computation of a locus specified with the i2g coding. These and other examples are available at `http://webs.uvigo.es/fbotana/THedu11/` together with some demo videos.

3.1 A GeoGebra example of proof

We consider the basic theorem that states that the three altitudes of a triangle meet in one point, its orthocenter. This result can be easily reproduced in terms of a boolean variable in a GeoGebra construction. Given triangle ABC, it suffices to consider the intersection point P of the altitudes a and b (through vertices A and B respectively). The boolean statement encompassing the theorem is $line(C,P) \perp line(A,B)$. The *true* provided by GeoGebra based on numerical computations is symbolically corroborated by Sage.

Notice that, unlike the answer provided by GeoGebra, the answer provided by Sage is not only based on symbolic computations but also completely general. Symbolic variables (u_1 to u_6) are used as generic coordinates for the three vertices, what makes the answer a general statement about any generic triangle.

3.2 An Intergeo example of locus

To obtain a locus description, one can use any of the DGS supporting the i2g format (see [1] for a list of softwares and their current implementation status). We consider the example provided in the Intergeo web site as an example of a construction with the `locus_defined_by_point_on_circle` element. It is the construction, in JSXGraph, of a cardioid as the locus set traced by a point P as a point X runs along a circle [14]. In fact, the i2g description provided by JSXGraph had a doble definition for the intersection point X that had to be changed elliminating its definition as a free point.

Figure 1 shows the equation and graph of the locus provided by Sage.

Acknowledgements

The authors thank Rado Kirov for his helpful indications on Sage programming and acknowledge the financial support by research grant MTM2008-04699-C03-03/MTM from the Spanish MICINN.

References

[1] *Intergeo Implementation Table*. Available at `http://i2geo.net/xwiki/bin/view/I2GFormat/ImplementationsTable`.

[2] F. Botana: *A Symbolic Companion for Interactive Geometric Systems*. To be presented at Conference on Intelligent Computer Mathematics (CICM 2011).

[3] F. Botana & J. L. Valcarce (2002): *A dynamic-symbolic interface for geometric theorem discovery*. *Computers and Education* 38, pp. 21–35.

[4] F. Botana & J. L. Valcarce (2003): *A Software Tool for the Investigation of Plane Loci*. *Math. Comput. Simul.* 61(2), pp. 139–152.

[5] GeoGebra Google Summer of Code: Available at `http://www.geogebra.org/trac/wiki/LocusLineEquation`.

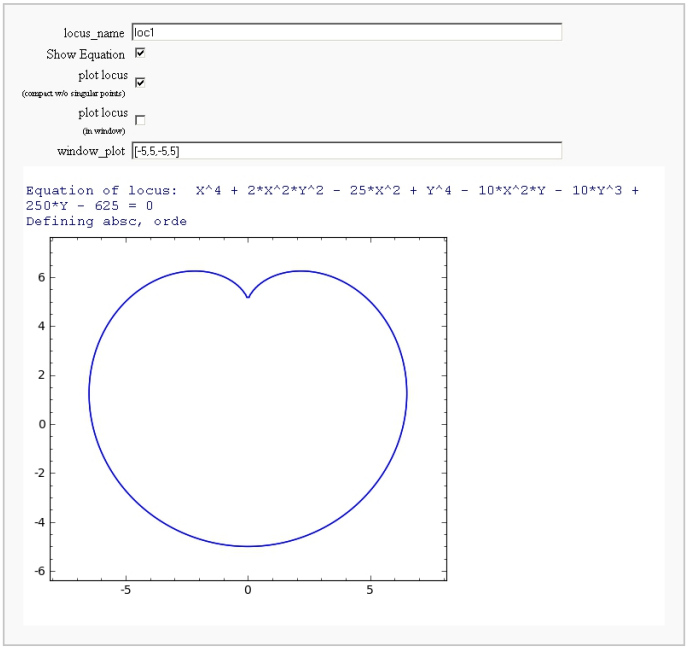


Figure 1: Equation and graph of cardioid as given by Sage.

[6] J. Escribano, F. Botana & M. A. Abnades (2010): *Adding Remote Computational Capabilities to Dynamic Geometry Systems*. *Mathematics and Computers in Simulation* 80, pp. 1177–1184.

[7] Intergeo file format: Available at `http://i2geo.net/xwiki/bin/view/I2GFormat/`.

[8] X. S. Gao, J. Z. Zhang & S. C. Chou (1998): *Geometry Expert*. Nine Chapters, Taiwan.

[9] GeoGebra: Available at `http://www.geogebra.org`.

[10] C. Hoyles & K. Jones (1998): *Proof in Dynamic Geometry Contexts*, chapter Perspectives on the teaching of Geometry for the 21st Century, pp. 121–128. Kluwer, Dordrecht.

[11] GeoGebra International Institutes: Available at `http://www.geogebra.org/igi`.

[12] Intergeo: Available at `i2geo.net/`.

[13] JSXGraph: Available at `http://jsxgraph.org/`.

[14] JSXGraph: *i2g locus example*. Available at `http://i2geo.net/xwiki/bin/download/I2GFormat/WebHome/locuscardioid.html`.

[15] U. Kortenkamp, A. M. Blessing, C. Dohrmann, Y. Kreis, P. Libbrecht & C. Mercat (2009): *Interoperable interactive geometry for europe – first technological and educational results and future challenges of the intergeo project*. In: *Proceedings of CERME 6, Lyon.*

[16] U. Kortenkamp, C. Dohrmann, Y. Kreis, C. Dording, P. Libbrecht & C. Mercat (2009): *Using the intergeo platform for teaching and research*. In: *Proceedings of the 9th International Conference on Technology in Mathematics Teaching, Metz, ICTMT-9.*

[17] J. Richter-Gebert & U. Kortenkamp (1999): *The Interactive Geometry Software Cinderella*. Springer, Berlin.

[18] E. Roanes-Lozano, E. Roanes-Macías & M. Villar (2003): *A bridge between dynamic geometry and computer algebra*. *Mathematical and Computer Modelling* 37, pp. 1005–1028.

[19] W. Stein: Available at `http://wstein.org/`.

[20] D. Wang (1996): *Automated Deduction - Cade-13*, chapter GEOTHER: A geometry theorem prover, pp. 166–170. *LNCS* 1104/1996, Springer.

Formalized Computational Origami in Education
Extended Abstract

Cezary Kaliszyk
Symbolic Computation Research Group
University of Tsukuba, Japan
kaliszyk@cs.tsukuba.ac.jp

In this paper we discuss using formalized origami in education. Origami allow introducing students to various mathematical problems in an attractive way. This has usually been done only on paper. Here we propose using computational origami from computer algebra and formalized origami to allow students interactively prove properties of the performed constructions also with the help of the computer.

1 Introduction

Origami is the traditional Japanese art of paper folding. The term is commonly associated with folding paper by hand. However, it is not restricted to an art. It has applications in science and technology as well as in education; this is a reason why it is becoming more popular in the recent years. Origami can be a tool for geometrical constructions; namely it can represent objects using paper folds. Instead of ruler and compass in Euclidean geometry one can use paper folds as a basis to create new points and lines on a surface.

Computational origami is a scientific discipline for studying mathematical and computational properties of *origami* (paper folding). It studies mathematical theories of paper folding, modeling origami by algebraic and symbolic methods, as well as computer simulation of paper folding and proving properties of the constructed origami by geometric and algebraic methods. Recently a system for computational origami EOS has been created [9]. This system is capable of visualizing origami constructions based on Huzita's axioms, analysing the origami folds algebraically, and showing properties of the constructions.

There is also an ongoing effort to formalize origami proofs [6]. This includes formalizing the origami fold principles and providing tactics for dealing with proof obligations arising from the origami proofs.

In this paper we discuss how a formal system for origami can be used in education. We combine features from existing components to create an environment for students to look at origami constructions not only using hand folding and traditional proofs, but also with the help of a proof assistant.

1.1 Contents

The rest of this paper is organized as follows. In Section 3 we describe ProofWeb, a system for proof assistants in education. In Section 2 talk about origami in education and proving properties in computational origami. Finally in Section 4 we give a conclusion and present possible future work.

2 Computational Origami

Computational origami is based on a set of basic fold operations proposed by Huzita and Justin [3, 4, 5]. They have classified all the possible folds, creating a system of operations for origami geometry (called

Submitted to:
THedu'11

axioms in the literature). The system has also been shown to be complete; which means that there are no other possible folds that superpose points and lines (for the case of folding along one line). It is known that the set of folds proposed by Huzita, is more powerful than Euclidean tools, i.e. straightedge and compass. Therefore, the class of the constructions possible is richer than that of Euclidean geometry. For example trisector of a given arbitrary angle is constructable by origami.

Origami has also been tried in some high schools in Japan [**?**] in introductory mathematical courses. Instead of ruler and compass in Euclidean geometry one can use paper folding to create new points and lines on a surface. Starting with a square area, one can specify folds to superpose existing points or lines. The creases obtained by such superpositions create new lines and the intersections of the new lines with existing ones create new points on the origami. Students are first given a real square origami paper and some simple tasks, like dividing a segment into equal sub-segments.

Later more complicated constructions are shown, in particular ones that can be obtained with origami but cannot be performed with ruler and compass. Students are shown the Abe's method of trisection of an angle and are shown simple theorems, like the Haga theorem [1]. These can be related to constructions that cannot be performed with Euclidean geometry. Students in Japan have been exposed to origami culturally; however most of them do not know the relation of origami and mathematics. In the past few years this has met a big interest in the students.

3 ProofWeb

Proof assistants are becoming more important tools in various domains of computer science and mathematics. However in computer science curricula they are presented only rarely and to students of higher years. Even further, most mathematics curricula never present proof assistants.

One tool that has been developed in the recent years is ProofWeb [2], a web interface to proof assistants designed especially with education in mind. In ProofWeb gives an interface to the Coq proof assistant, that has been additionally extended with special proof tactics and proof display. This allows using a proof assistant in introductory logic courses. Additional tactics allow the students to reason in certain simple logical systems and the special display renders proof trees in various natural deduction styles [8].

ProofWeb has been made available publically, and as such has been used by in a number of courses. More than 20 courses and 800 students have used our server [7] and we helped set up at least three other servers running ProofWeb at different universities. The courses range over subjects like:

- Introductory courses to logic. Students are taught basic logical systems; ProofWeb already includes a database of more than a hundred exercises in propositional logic and predicate logic.
- Courses about modelling in logic. Students are modelling simple linear systems, like a water boiler.
- Type theory courses based on the Coq proof assistant
- Proof assistants courses
- Derivation of algorithms. Students are shown extraction of programs from proof assistant formalized code.
- Semantics of programming languages.

Currently the system has not been used for proofs of geometrical statements. In fact using proof assistants for geometry is usually cumbersome because of the relations between the display and the representation of the objects for the proof assistant.

We have expressed the origami fold axioms as formal statements and included them as a ProofWeb library. We proided a simplified interface for students to combine folds. This allows enriching an introduction to mathematics course that includes geometry methods by allowing formally proving properties of the constructions.

4 Conclusion

Studying origami problems within math courses has given positive results in the past. In this paper we have discussed how this can be combined with computer technology. Origami axioms have been expressed as part of the interactive origami system Eos. When properties are proven with Eos, the resulting proof document shows a combination of deduction steps and computation performed within computer algebra.

Origami axioms expressed formally in a proof assistant are made available to the students as a library (module) for ProofWeb. This extends the domains where ProofWeb can be applied in education to origami geometry. The students do not need to install any software, as the interface is provided over the web. Origami problems expressed formally can be given to students in the form of ProofWeb tasks; and origami operations in form of tactics.

The created approach is an integration of techniques from various domains of computer science, like computer algebra, computational geometry and proof assistants. Such combinations can be useful in creating future unified mechanized mathematical assistant tools, which would contribute not only to science but also to education.

4.1 Future Work

There are a number of ways in which the current system can be extended, most important for education are:

- Eos includes a mechanism for rendering origami on the Web. A similar mechanism could be added to ProofWeb to allow interactively viewing the construction being analysed.
- More proofs about origami constructions can be proved to be presented to the students inside the formal system.
- A more detailed comparison with dynamic geometry systems, commonly used in education could be performed.
- Proof documents that accompany Eos proofs are made to be human readable. Similarly proofs in logic done with ProofWeb are accompanied by the deduction trees in various styles [8]. A mechanism for presenting the proofs in a human readable way would certainly make it more attractive for mathematics.

Finally the system has not yet been tried in a complete course. Receiving feedback would allow for more evaluation of the application of computers in teaching mathematics with origami.

References

[1] K. Haga (1999): *Origamics Part I: Fold a Square Piece of Paper and Make Geometrical Figures (in Japanese)*. Nihon Hyoronsha.

[2] Maxim Hendriks, Cezary Kaliszyk, Femke van Raamsdonk & Freek Wiedijk (2010): *Teaching logic using a state-of-the-art proof assistant*. Acta Didactica Napocensia 3(2), pp. 35–48.

[3] H. Huzita (1989): *Axiomatic Development of Origami Geometry*. In H. Huzita, editor: Proceedings of the First International Meeting of Origami Science and Technology, pp. 143–158.

[4] H. Huzita (1989): *The Trisection of a Given Angle Solved by the Geometry of Origami*. In H. Huzita, editor: Proceedings of the First International Meeting of Origami Science and Technology, pp. 195–214.

[5] Jacques Justin (1989): *Résolution par le pliage de l'équation du troisième degré et applications géométriques*. In Humiaki Huzita, editor: Proceedings of the First International Meeting of Origami Science and Technology, pp. 251–261.

[6] Cezary Kaliszyk & Tetsuo Ida: *Proof assistant decision procedures for formalizing origami*. Submitted to Calculemus 2011.

[7] Cezary Kaliszyk, Femke van Raamsdonk, Freek Wiedijk, Hanno Wupper, Maxim Hendriks & Roel de Vrijer (2008): *Deduction using the ProofWeb system*. Technical Report ICIS–R08016, Radboud University Nijmegen.

[8] Cezary Kaliszyk & Freek Wiedijk (2008): *Merging procedural and declarative proof*. In Stefano Berardi, Ferruccio Damiani & Ugo de'Liguoro, editors: *TYPES*, *LNCS* 5497, Springer, pp. 203–219.

[9] A. Kasem, T. Ida, H. Takahashi, M. Marin & F. Ghourabi (2006): *E-Origami System* Eos. In: Proceedings of the Annual Symposium of Japan Society for Software Science and Technology, JSSST, Tokyo, Japan.

Packed views of pre-structured data

Vladimir Komendantsky*
School of Computer Science
University of St Andrews
St Andrews
KY16 9SX, UK
vk10@st-andrews.ac.uk

We propose a technique of packed view for interactive communication between heterogeneous software such as theorem provers and computer algebra systems. A packed view is a method to infer, with a possible share of user interaction, the type-theoretic meaning of a given pre-structured object, as well as a method to display the underlying syntactic structure of a semantic term as a pre-structured object (essentially, an abstract syntax tree) corresponding to it. With this notion of view in hand, it is relatively straightforward to program non-trivial parts of communication interfaces between a theorem prover and a computer algebra using the logic of the prover as the programming language.

1 Introduction

In scientific as well as in educational applications there is a common problem of communication between different components of heterogeneous software. For example, we may consider communication links between two theorem provers, in which case we are likely interested in communicating a proof goal and its context, or a theory while preserving as much logic contents of terms as possible. A rather different kind of situation arises when a theorem prover is communicating with a non-logic based computational system. Such systems operate according to their own, possibly informal semantics, and may accept communications in a structure-oriented format based on the XML technology. A circumstance that we can use to our benefit is that the type of communicated structure depends on the domain of application of a given program rather on particularities of implementation of that program, such as operational semantics, whether it is formal or not. Therefore a semi-formal specification of the communication format should exist. Existence of a specification means that the type of communicated structure, or at least its faithful subpart, can be presented as an inductive datatype in a prover.

Given an inductive datatype representing the class of communicated abstract syntax trees (ASTs) recognised by the prover, we can think about an embedding of these ASTs into the logic of the prover. For such an embedding, we can consider theories or type-theoretic hierarchies of mathematical objects. It is sensible to allow this embedding to be partial since not all syntactic expressions have a meaning and therefore not all possible ASTs can be rendered as objects in the corresponding theory.

The inductive type of AST is our type of *pre-structured data*. Pre-structuring can be done at the level of implementation language of the prover. It essentially consists of taking a representation of an AST in an XML-based format as input, parsing it, and producing a well-typed term of the prover's type of AST, and vice versa. All these operations are rather straightforward and only require some input/output capabilities. The most interesting and important part is to assign meaning to ASTs in the type theory of the prover by defining functions that take ASTs as input and construct objects in the respecting theory.

*Supported by the research fellowship EU FP7 Marie Curie IEF 253162 'SImPL'.

Submitted to:
THedu'11

At the pre-structuring stage, nothing is known about the meaning of a given AST, even whether such a meaning exists at all. By finding a meaning for the AST, we view the syntactic object corresponding to it as a semantic object. Assignment of a meaning to an object is done by a semantic interpretation function. The converse process performed by a display function that translates a semantic object to the syntactic form for the purpose of displaying it either to a user or to another program. The former process infers the semantic interpretation of an object, while the latter process forgets it.

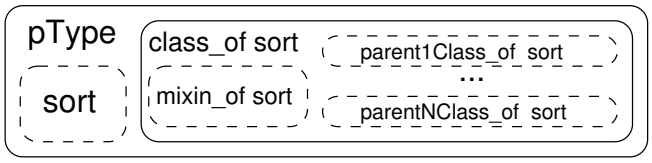
Contribution. We introduce a notion of packed view of a pre-structured object. A packed view is a method to infer, with a significant share of user interaction, the type-theoretic meaning of a given pre-structured object, as well as a method to display the underlying syntactic structure of a semantic term as a pre-structured object (essentially, an abstract syntax tree) corresponding to it. With this notion of view in hand, it is relatively straightforward to program non-trivial parts of communication interfaces between a theorem prover and a computer algebra system inside the theorem prover itself, as an alternative to having a complicated and difficult to extend interface at the level of implementation language of the theorem prover.

Outline. This short paper discusses generic packed views in Section 2. An example of pre-structured data, a subtype of the OpenMath standard, is given in Section 3. Our conclusions are contained in Section 4.

Notational conventions. Our meta-language, Coq [10], has dependent products of the form $\forall\ (x : A).\ B$ where x is a variable which is bound in B; the case when x is not free in B is denoted $A \to B$ which is a simple, non-dependent type. Also, Coq features inductive and coinductive type definitions. For the sake of presentation, we do not provide listings of Coq code in Sec. 2. We use a human-oriented type-theoretic notation, where $\star$ denotes the predicative universe of types, Type, and inductive and coinductive definitions are displayed in natural-deduction style with single and, respectively, double lines.

Related work. Views were considered by Wadler in [11] as a method to convert from concrete data with free structure to abstract data. The problem was to provide a formal concept of definitional correspondence between structured data that allows pattern matching and data with any structure being missing, e.g., for the purpose of efficiency. The user of a programming language supporting views receives a method to access features specific either to concrete or to abstract data (resp., pattern matching and representation hiding) working with one and the same object by viewing that object as either concrete or abstract.

Our method is based on the design pattern of *packed types* introduced in [3]. A packed type pType can be depicted as follows, in an object-oriented programming style:



Enclosed in boxes are types. Dashed borders indicate a type to which the containing type can be coerced. A *packed class* is a container inductive structure supplied with pack and unpack operations that, together with unification hints implemented in Coq (that is, implicit coercions and canonical structures), allows to be organised in a consistent and extendible hierarchy of structures and has been employed in [3, 5] to define a mathematical type hierarchy. The authors of [3] mentioned that, unlike the straightforward telescopic model of inheritance, packed types allow natural multiple inheritance and a reduced size of terms thanks to putting separate components of mathematical structures in records called *mixins*, and reducing structure nesting depth in general. The latter is crucial as a workaround to cope with the type inference algorithm of Coq that is exponential in the size of term which, if hierarchies are concerned,

should be of the order C^n, where C is the number of the components of the structure, and n is the structure nesting depth. There is also a current effort to apply packed types to theorem proving automation in [4].

The problem of finding a meaning of pre-structured data in Coq was previously approached in the context of requesting computer algebra system computations from the prover in [8, 7]. Other approaches to computer algebra are also practised in the theorem proving community. There are two major related trends which however have quite different aims to ours: 1) building a computer algebra system on top of a proof assistant [6, 1], and 2) creating a programing environment for development of certified computation [2]. Unlike any of these developments, we do not approach the nature of computational algorithms but rather concern ourselves with the task of representing given computer algebra data in a way acceptable in a theorem prover.

2 A view

The base type of view. *Mixins* are introduced as elementary building blocks for containers such as viewType below. A mixin is an inductive type together with projections to its constructors. Below is the base view mixin for a type of pre-structured data called OM:

$$\mathsf{mixin\_of} : \star \to \star$$

$$\frac{\mathit{viewin} : \mathsf{OM} \to \mathsf{option}\ \mathit{iT} \qquad \mathit{viewout} : \mathit{iT} \to \mathsf{option}\ \mathsf{OM}}{\mathsf{ViewMixin}\ \mathit{viewin}\ \mathit{viewout} : \mathsf{viewMixin\_of}\ \mathit{iT}}$$

Here, option is the polymorphic type of fan ordering with values either of the kind Some of a value of a given type, or None, with the latter representing the absence of translation. Field projections are the following:

$$\begin{aligned} \mathsf{viewin} &= \lambda\ (\mathit{iT} : \star)\ (m : \mathsf{viewMixin\_of}\ \mathit{iT}).\ \mathbf{let}\ (\mathit{viewin},\ \_) := m\ \mathbf{in}\ \mathit{viewin} \\ \mathsf{viewout} &= \lambda\ (\mathit{iT} : \star)\ (m : \mathsf{viewMixin\_of}\ \mathit{iT}).\ \mathbf{let}\ (\_,\ \mathit{viewout}) := m\ \mathbf{in}\ \mathit{viewout} \end{aligned}$$

The *class* of the base type is simply its mixin.

$$\mathsf{viewClass\_of} = \mathsf{viewMixin\_of}$$

In the packed type methodology, containers $*$type pack a given representation type with given base classes and the underlying mixin. In the base case, we have only a mixin. The container viewType is given below. The third argument U is required for the purpose of unification.

$$\mathsf{type} : \star$$

$$\frac{\dfrac{\mathit{viewSort} : \star}{\mathit{viewClass} : \mathsf{viewClass\_of}\ \mathit{viewSort}} \qquad U : \star}{\mathsf{ViewPack}\ \mathit{viewSort}\ \mathit{viewClass}\ U : \mathsf{viewType}}$$

Now we have to provide a hint for the unification algorithm that allows to coerce viewType to its representation type. This function is delegated to the field projection viewSort.

$$\mathsf{viewSort} = \lambda\ (t : \mathsf{viewType}).\ \mathbf{let}\ (\mathit{viewSort},\ \_,\ \_) := t\ \mathbf{in}\ \mathit{viewSort}$$

The second field projection, viewClass, is not associated with a coercion.

$$\begin{aligned} \mathsf{viewClass} &: \forall\ \mathit{cT} : \mathsf{viewType}.\ \mathsf{viewClass\_of}\ (\mathsf{viewSort}\ \mathit{cT}) \\ \mathsf{viewClass}\ \mathit{cT} &= \mathbf{let}\ \mathsf{ViewPack}\ \_\ c\ \_ := \mathit{cT}\ \mathbf{in}\ c \end{aligned}$$

Finally, we define the constructor ViewType for the type viewType, and input and output view functions:

$$\begin{aligned} \mathsf{ViewType}\ T\ m &= \mathsf{ViewPack}\ T\ m\ T \\ \mathsf{In} &= \lambda\ (iT : \mathsf{viewType}).\ \mathsf{viewin}\ iT\ (\mathsf{viewClass}\ iT) \\ \mathsf{Out} &= \lambda\ (iT : \mathsf{viewType}).\ \mathsf{viewout}\ iT\ (\mathsf{viewClass}\ iT) \end{aligned}$$

Higher-level types. The base type of view can be extended, for example, with a notion of correctness of a view for the purpose of certification of computer algebra computations, or by adding other kinds of conversion functions for extended usability. Construction of view types given other view types is reminiscent of construction of objects in object-oriented programming, and can be depicted diagrammatically. A generic diagram is shown in Introduction. The main difference with the base type of view is in the class of the higher-level type that can contain a number of mixins. In the extended class, the respective field projections coerce it to the base classes and the underlying mixin but *not* to the representation type iT, which facilitates multiple inheritance.

3 Instantiated views

In this section we consider a concrete type of semi-structured data. This is a datatype called OM representing a certain subset of the OpenMath standard[9]. The definition is by induction as follows:

$$\mathsf{OM} : \star$$

$$\frac{x : \mathbb{Z}}{\mathsf{OMInt}\ x : \mathsf{OM}} \qquad \frac{s : \mathsf{string}}{\mathsf{OMVar}\ s : \mathsf{OM}} \qquad \frac{s_1\ s_2 : \mathsf{string}}{\mathsf{OMSym}\ s_1\ s_2 : \mathsf{OM}} \qquad \frac{o : \mathsf{OM} \quad os : \mathsf{list\ OM}}{\mathsf{OMApp}\ o\ os : \mathsf{OM}}$$

The above definition can be programmed as an inductive datatype in Coq:

```
Inductive OM : Type := OMInt : Z → OM | OMVar : string → OM
  | OMSym : string → string → OM | OMApp : OM → list OM → OM.
```

Polymorphism. Given a canonical structure T : viewType of view, we can construct polymorphic views. For example, a view for polymorphic lists can be constructed below. The *canonical structure* list_viewType is a unification hint that allows to unify function arguments of type list T with the view of lists and hence construct views by unification.

```
Fixpoint om_list : OM → option (list T) := ...
Fixpoint list_om : list T → option OM := ...
Definition list_viewMixin := ViewMixin _ om_list list_om.
Canonical Structure list_viewType := Eval hnf in ViewType (list T) list_viewMixin.
```

Dependent types. Dependent types can be treated using the inductive type of dependent pairs in Coq, which allows to delay proof construction until the stage of interactive proof. For example, one of possible representations of polynomials (a list of coefficients with a proof that the last element is not 0, unless the list is empty) can be treated as follows, again, for a given T : viewType. For this, however, we also provide a mapping from the type poly T of polynomial over T to the corresponding dependent pair type for the view to be useful in a context.

```
Fixpoint om_poly : OM → option {c : list T & last 1 c ≠ 0 → poly T} := ...
Fixpoint poly_om : {c : list T & last 1 c ≠ 0 → poly T} → option OM := ...
Definition poly_viewMixin := ViewMixin _ om_poly poly_om.
Canonical Structure poly_viewType := Eval hnf in ViewType _ poly_viewMixin.
```

4 Conclusions

Views can be constructed piecewise by providing views for 1) simple types, 2) polymorphic types and 3) dependent types, and then providing hints for the unification algorithm of the theorem prover to infer the type of view given the type of argument of functions In and Out that perform viewing.

The views paradigm can be especially useful in the prototype Coq-to-GAP communication interface currently available for download at `http://www.cs.st-andrews.ac.uk/~vk/Coq+GAP/`. One of possible projects can be connected with visualisation of the object-oriented structure of packed types outside the theorem prover, and interacting with the communication between the prover and the computer algebra tool by means of graphical construction of views.

References

[1] Yves Bertot, Frédérique Guilhot & Assia Mahboubi (2011): *A formal study of Bernstein coefficients and polynomials*. *Mathematical Structures in Computer Science* Available at `http://hal.inria.fr/inria-00503017/en/`.

[2] S. Boulmé, T. Hardin, D. Hirschkoff, V. Ménissier-Morain & R. Rioboo (1999): *On the way to certify Computer Algebra Systems*. *Electronic Notes in Theoretical Computer Science* 23(3), pp. 370–385, doi:10.1016/S1571-0661(05)80609-7. CALCULEMUS 99, Systems for Integrated Computation and Deduction (associated to FLoC'99, the 1999 Federated Logic Conference).

[3] François Garillot, Georges Gonthier, Assia Mahboubi & Laurence Rideau (2009): *Packaging mathematical structures*. In: *Theorem Proving in Higher Order Logics (2009), LNCS* 5674. Available at `http://hal.inria.fr/inria-00368403_v2/en/`.

[4] G. Gonthier, B. Ziliani, A. Nanevski & D. Dreyer (2011): *How to Make Ad Hoc Proof Automation Less Ad Hoc*. Manuscript.

[5] Georges Gonthier, Assia Mahboubi & Enrico Tassi (2011): *A Small Scale Reflection Extension for the Coq system*. Research Report RR-6455, INRIA. Available at `http://hal.inria.fr/inria-00258384/en/`.

[6] Cezary Kaliszyk & Freek Wiedijk (2007): *Certified Computer Algebra on Top of an Interactive Theorem Prover*. In: *Proceedings of the 14th symposium on Towards Mechanized Mathematical Assistants: 6th International Conference*, Calculemus '07 / MKM '07, Springer-Verlag, Berlin, Heidelberg, pp. 94–105, doi:10.1007/978-3-540-73086-6_8.

[7] Vladimir Komendantsky, Alexander Konovalov & Steve Linton (To appear): *Interfacing Coq + Ssreflect with GAP*. In: *Proc. User Interfaces for Theorem Provers (UITP) 2010*, ENTCS, Elsevier.

[8] Vladimir Komendantsky, Alexander Konovalov & Steve Linton (To appear): *View of computer algebra data from Coq*. In: *Proc. Conference on Intelligent Computer Mathematics (CICM) 2011*, LNAI, Springer.

[9] OpenMath: `http://www.openmath.org/`.

[10] The Coq development team: *The Coq Proof Assistant Reference Manual*. `http://coq.inria.fr/refman/`.

[11] Philip Wadler (1987): *Views: A Way for Pattern Matching to Cohabit with Data Abstraction*. In: *POPL'87*, pp. 307–313, doi:10.1145/41625.41653.

Formalization and Implementation of Algebraic Methods in Geometry

Filip Marić, Predrag Janičić, Ivan Petrović, Danijela Petrović
Faculty of Mathematics,
University of Belgrade, Serbia
filip@matf.bg.ac.rs, janicic@matf.bg.ac.rs, mr00006@alas.matf.bg.ac.rs, danijela@matf.bg.ac.rs

We describe our ongoing project of formalization of algebraic methods for geometry theorem proving (Wu's method and Gröbner bases method) and their integration in dynamic geometry tools widely used in education. The project includes formal verification of the algebraic methods within Isabelle/HOL proof assistant and development of a new, open-source Java implementation of the algebraic methods. The project should fill-in some gaps still existing in this area (e.g., lack of open-source implementations, lack of formal links between algebraic methods and synthetic geometry) and should enable new applications of the algebraic methods, especially in education.

1 Introduction

The field of automated deduction in geometry has been very successful in the last several decades and a number of *geometry automated theorem provers (GATP)* has been developed. Most successful of these are based on *algebraic methods*, primarily Wu's method [21] and the method of Gröbner bases [2]. Algebraic methods require expressing geometric properties as polynomial equations in the Cartesian plane, and then using algebraic techniques for dealing with these equations. There are many implementations of these methods and hundreds of complex geometry theorems has been proved automatically by them. However, despite these advances, there are still some gaps in this area, preventing wider applications of the algebraic methods for geometry, especially in education and in formal explorations of geometry. Some of these gaps are:

- There are no free, open source, and well-documented implementations of GATPs based on algebraic methods, suitable for integration with other tools (e.g., dynamic geometry tools).
- There is no support for automated theorem proving in dynamic geometry tools most widely used on all levels of mathematical education (e.g., GeoGebra [9]) and this limits their applicability.
- Only fragments of geometry and algebraic methods have been formalized within proof assistants (e.g., Isabelle, Coq), and there are still no formalized links between algebraic methods and synthetic geometry, giving formal correctness arguments for algebraic methods for geometry.
- There are still no standardized interchange formats for geometry theorems (although there are initiatives for defining them) and state-of-the art automated theorem provers use only their custom input languages.

We have recently started a project in which we focus on integrating algebraic methods in geometry with dynamic geometry (DG) tools (widely used in education) and in interactive proof assistants (used for formalizing geometry). One of our goals is to address formalization of algebraic methods and their integration into the interactive proof assistant Isabelle/HOL. A major concern when using algebraic methods is the lack of formally established connections between proofs produced by these methods with classical

Submitted to:
THedu'11

synthetic proofs in geometry (i.e., with Hilbert's or Tarski's geometry). We aim to make a strong, formal link between algebraic methods and synthetic geometry by formalizing both within Isabelle/HOL. Once the correctness of algebraic methods is formally established within a proof-assistant, these methods can be used as reflective tactics that help automating significant portions of proofs in exploring geometry in a formal setting. Our other goal is to develop Java implementations of the Wu's method and the Gröbner bases method conforming to open-source code and documentation standards. These should directly support current standardization initiatives for geometry formats (e.g., i2g, TGTP) and should be suitable for integration into DG tools. Both these goals, if achieved, would enable new applications of (both automated and interactive) theorem provers in education and in formalizing mathematics.

2 Formalization of Algebraic Methods within Isabelle/HOL

There is a number of formalizations of fragments of various geometries within proof assistants. Parts of Hilbert's seminal book *Foundations of Geometry* have been formalised in Isabelle/Isar [14] and in Coq [5]. Within Coq, there are also formalizations of von Plato's constructive geometry [12], French high school geometry [8], Tarski's geometry [16], ruler and compass geometry [6], projective geometry [13], etc. There are efforts to integrate Gröbner bases solvers with proof-assistants [17, 3], but only the general part, not the one dealing with geometry statements. The link with synthetic geometry can be reconstructed from the mathematical literature, but, as far as we know, it is still not formally established.

Our goals are:

- to formally analyze the connections between different algebraic methods;
- to build formally verified geometry axiomatizations and models, and to implement algebraic GATPs that are formally verified, yet efficient enough to handle non-trivial geometric statements;
- to integrate algebraic GATPs to Isabelle/HOL and facilitate their use in education and in formal explorations in geometry.

In this section we present first steps and some choices made towards formalization of algebraic methods (Wu's method and Gröbner bases method) within the Isabelle/HOL proof assistant.

Term representation of geometric conjectures. Abstract syntax for representing geometric statements (of constructive type) within Isabelle/HOL is defined by mutually recursive datatypes `point_term` and `line_term` (corresponding to point and line constructions) and `statement_term` (corresponding to statements). In this syntax, for example, the statement that three perpendicular bisectors of edges of a triangle *ABC* meet in a single point can be represented by the following HOL term (points `A`, `B`, `C` are, so-called, *free points*):

```
let A = MkPoint 1; B = MkPoint 2; C = MkPoint 3;
    C1 = MkMidpoint A B; A1 = MkMidpoint B C; B1 = MkMidpoint C A;
    O = MkIntersection (MkNormal (MKLine A B) C1) (MkNormal (Mkline B C) A1) in
    Incident O (MkNormal (Mkline C A) B1)
```

Interpreting terms in geometry models. Syntactic terms that represent geometry statements can be given various semantics by interpreting in different models of geometry (e.g., Cartesian plane, Hilbert's geometry, Tarski's geometry). Isabelle's *locales* infrastructure is used to avoid repeating definitions. A locale `AbstractGeometry` is defined that contains primitive relations needed to interpret a geometric statement (e.g., *incident*, *between*, *congruent*), and various models of geometry interpret this locale.

Semantics of a term (in abstract geometry) is given by the functions `point_interp`, `line_interp` and `statement_interp`, that take a `point_term`, `line_term` or a `statement_term` and return a (abstract) point, a (abstract) line or a Boolean value, respectively. Since abstract interpretation of a term is uniquely determined only if free points are fixed, all these functions take an additional argument — a function mapping free point indices to points.

Statements are interpreted by the primitive relations of the abstract geometry, while constructions are reduced to primitive relations by use of Hilbert's ε operator (`SOME` in Isabelle/HOL). For example:

```
statement_interp (Incident p l) fp =
   incident (point_interp p fp) (line_interp l fp)
point_interp (MkIntersection l1 l2) fp =
   (SOME P. incident P (line_interp l1 fp) & incident P (line_interp l2 fp))
```

A statement (represented by a term) is valid in an (abstract) geometry if all its interpretations (i.e., interpretations for all choices of free points) are true.

```
definition (in AbstractGeometry) valid :: "statement_term => bool" where
  "valid stmt = (ALL fp. statement_interp stmt fp)"
```

All the previous notions lift to concrete geometry models (e.g., Cartesian plane, Hilbert's geometry, Tarski's geometry) once it is shown that these are interpretations of the `AbstractGeometry` locale.

Formalizing analytic geometry. A Cartesian model of Euclidean geometry has been defined within Isabelle/HOL. Points are defined as pairs of real numbers. Lines are determined by their equations of the form $Ax+By+C=0$, so a line is defined as an equivalence class of triplets (A,B,C) where $A \neq 0 \vee B \neq 0$ and triplets are equivalent iff they are proportional. This kind of reasoning has been facilitated by support for quotient types and quotient definitions that has been recently introduced to Isabelle/HOL.

Once the types of points and lines are introduced, they are used to interpret the `AbstractGeometry` locale by introducing primitive geometric predicates and proving their properties (basically axiom-level statements). For example, the predicate *incidenct* is defined by a quotient definition, stating that a point (x,y) is incident to a line containing coefficients (A,B,C) iff $A \cdot x + B \cdot y + c = 0$, and for example, it is proved that there is a unique line incident to two different points.

Algebrizing geometric terms. Before algebraic methods can be applied, a geometric statement must be expressed in an algebraic form (as a set of polynomial equations, or just polynomials, assuming equations are always of the form $p(x) = 0$). Standard algebrization procedure introduces fresh symbolic variables for unknown point coordinates and introduces polynomial equations that characterize every construction step and the conjecture to be proved. We have implemented such a procedure within Isabelle/HOL. It keeps track of the current state containing symbolic coordinates of points introduced during the construction (these are subterms of the original statement term) and polynomials (both for construction and the statement) obtained so far.

The function `algebrize` takes a statement term and returns sets of construction and statement polynomials obtained by our algebrization procedure, starting from an empty initial state. The procedure descends the term recursively and adds specific polynomials for each construction step or for the statement. For example, when algebrizing a statement of the form `Colinear A B C`, the set of statement polynomials includes the polynomial $(x_a - x_b) \cdot (y_b - y_c) - (y_a - y_b) \cdot (x_b - x_c)$, where (x_a, y_a), (x_b, y_b), and (x_c, y_c) are symbolic coordinates introduced during algebrization of the constructions for point terms

A, B, and C (during this, the set of construction polynomials was appropriately extended so that it contains definitions of these symbolic coordinates in term of symbolic coordinates of free points).

The central theorem that we have formally proved is that, if all statement polynomials are zero whenever all construction polynomials are (i.e., if all statement polynomials belong to the ideal generated by the construction polynomials), then the statement is valid in analytic geometry.

```
theorem "let (cp, sp) = algebrize term in
(ALL ass. ((ALL p : cp. eval_poly ass p = 0) --> (ALL p : sp. eval_poly ass p = 0)) -->
     AnalyticGeometry.valid s)"
```

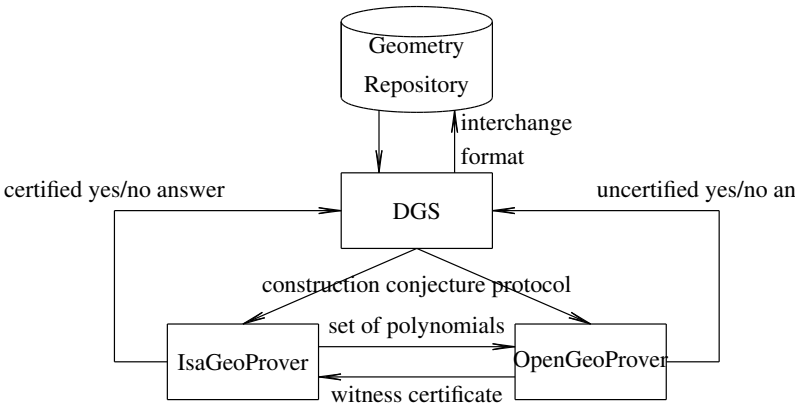
Connections with synthetic geometry. Our further goals include formalizing the connections between algebraic methods and synthetic axiomatizations (connections can be established via analytic geometry of the Cartesian plane). The first step would be to formalize Tarski's elementary geometry within Isabelle/HOL. Then, it would be necessary to prove that every valid statement in analytic geometry also hold in all models of Tarski's geometry. The central (most demanding) part would then be to formally prove the deductive completeness of Tarski's geometry (by formally establishing the connection with real-closed fields and formally showing that they allow quantifier-elimination). Next step would be to analyze the connections with Hilbert's geometry by showing that all proofs in Tarski's geometry can be transferred to Hilbert's geometry (since all Tarski's axioms can be proved within Hilbert's geometry). Finally, if an algebraic method is proved to be correct (in purely algebraic terms), then it would follow that it is correct when used for geometry theorem proving.

Non-degeneracy conditions. Very important feature of algebraic methods is that they automatically obtain conditions under which the given statement holds (so called, non-degeneracy or NDG conditions). In the early stages of our project we intend to use Groöbner basis solvers already integrated into Isabelle/HOL that are not capable of finding the required NDG conditions, so we plan to allow users to either specify NDGs or use the most general NDGs that can be inferred from the construction description. In the later stages we plan to integrate the Wu's method with Isabelle/HOL and then it would be possible to obtain the necessary NDGs automatically and handle them in an appropriate way.

3 Open Implementation of Algebraic Methods

Just a few dynamic geometry tools have support for automated theorem proving, typically based on Wu's method, the Gröbner bases method, and the area method. Some of them are *GEOTHER* [19], *MMP/Geometer* [7], *Discover* [1], *Geometry Explorer* [20], *Geometry Expert*/*Java Geometry Expert* [4, 22], *GeoProof* [15] (probably the only tool that combines dynamic geometry tool with automated and interactive theorem proving), *GCLC* [10]. Moreover, none of these provers is freely available as open-source (with exception of Narboux's implementation of the area method within Coq). Since implementing geometry theorem provers is a very demanding task, this lack of available implementations prevent wider use of provers in dynamic geometry tools.

We are developing a *OpenGeoProver* software (in Java) supporting Wu's method and Gröbner bases method for geometry. It is an improved reimplementation of the algebraic methods implemented in C++ within the *GCLC* tool [18]. OpenGeoProver consists of two main modules: one is the GATP based on these algebraic methods and the other is a set of various APIs from different geometric applications and formats to the prover. Internally, constructions are stored in a *Construction Conjecture Protocol (CCP)* format, inspired by GeoGebra, but containing the geometric statement that is being proved.



Integration of DGS, Isabelle geometry prover and OpenGeoProver

4 Big Picture

The components outlined above could and should be integrated with other tools and resources, as illustrated in Figure 4. A dynamic geometry tool can enable user to enter geometry constructions and conjectures or to import them from some repository (e.g., GeoThms[1]). The user should also be able to export his/her construction to the repository, and this communication should be performed using some of the emerging standard interchange formats (e.g., i2g). The dynamic geometry tool can invoke automated theorem prover on demand (to check if some conjecture is indeed a theorem), or to check if some construction is regular [11]. In either case, a conjecture, in the form of construction conjecture protocol, is sent to Isabelle (if a verified answer is required) or to OpenGeoProver (if efficiency is more important than a verified answer). If invoked, our geometry prover in Isabelle calls OpenGeoProver (or some other algebraic tool) and asks for a witness certificate (e.g., a Gröbner basis) that can be used for verified proving of the given conjecture. Finally, the yes/no answer and NDGs are returned by Isabelle prover or OpenGeoProver back to DGS. This interaction can bring new features to existing DGS and other related systems. For instance, compared to *GeoProof*, the proposed system would also provide efficient Java provers, while, compared to *GCLC*, the proposed system would provide link to a proof assistant and verified reasoning.

The proposed system with described interaction can be relevant for new applications in education. However, its components can be also useful in formal explorations of geometry. Namely, we are planning to implement an Isabelle tactics `wu` and `gb` that would be helpful for many intermediate tasks along building a formalized body of geometry knowledge.

5 Conclusions

In this note we briefly presented our ongoing project on automatizaton and formalization of algebraic methods for geometry theorem proving. The project should hopefully help wider applications of the algebraic methods in education and formalization of mathematics. Nevertheless, the real progress in this area can be expected once the efforts of all researchers involved are gathered through integrative projects, interchange formats, and shared implementations and formalizations.

[1] `http://hilbert.mat.uc.pt/~geothms/`

References

[1] Francisco Botana (2003): *A Web-Based Intelligent System for Geometric Discovery*. In: *International Conference on Computational Science*, *LNCS* 2657, Springer.

[2] B. Buchberger & F. (eds) Winkler (1998): *Gröbner Bases and Applications*. Cambridge University Press.

[3] A. Chaieb & Makarius Wenzel. Context aware Calculation and Deduction — Ring Equalities via Gröbner Bases in Isabelle. CALCULEMUS 2007 and MKM 2007. LNAI 4573. 2007.

[4] S.C. Chou, X.S. Gao & J.Z. Zhang (1996): *An Introduction to Geometry Expert*. In M. A. McRobbie & Slaney J. K., editors: *CADE 13*, *LNAI* 1104, Springer-Verlag.

[5] Christophe Dehlinger, Jean-François Dufourd & Pascal Schreck (2001): *Higher-Order Intuitionistic Formalization and Proofs in Hilbert's Elementary Geometry*. In: *Automated Deduction in Geometry*, *LNCS* 2061, Springer.

[6] Jean Duprat (2002): *Constructors: a ruler and a pair of compasses*. Types 2002.

[7] Xiao-Shan Gao & Qiang Lin (2004): *MMP/Geometer A Software Package for Automated Geometric Reasoning*. In Franz Winkler, editor: *Automated Deduction in Geometry: 4th International Workshop, (ADG 2002)*, *LNCS* 2930, Springer-Verlag.

[8] Frédérique Guilhot (2005): *Formalisation en* Coq *et visualisation d'un cours de géométrie pour le lycée*. *Revue des Sciences et Technologies de l'Information, Technique et Science Informatiques, Langages applicatifs* 24. Lavoisier.

[9] Markus Hohenwarter & Karl Fuchs (2004): *Combination of Dynamic Geometry, Algebra and Calculus in the Software System GeoGebra*. In: *Computer Algebra Systems and Dynamic Geometry Systems in Mathematics Teaching Conference 2004*.

[10] Predrag Janičić (2010): *Geometry Constructions Language*. *Journal of Automated Reasoning* 44(1-2).

[11] Predrag Janičić and Pedro Quaresma (2010): *Automatic Verification of Regular Constructions in Dynamic Geometry Systems*. *Automated Deduction in Geometry 2006* LNCS 4869.

[12] Gilles Kahn (1995): *Constructive Geometry according to Jan von Plato*. Coq contribution. Coq V5.10.

[13] Nicolas Magaud, Julien Narboux & Pascal Schreck (2008): *Formalizing Projective Plane Geometry in Coq*. In: *Proceedings of ADG'08*.

[14] Laura Meikle & Jacques Fleuriot (2003): *Formalizing Hilbert's Grundlagen in Isabelle/Isar*. In: *Proceedings of TPHOLs*, *LNCS* 2758, Springer.

[15] Julien Narboux (2007): *A Graphical User Interface for Formal Proofs in Geometry*. *Journal of Automated Reasoning* 39(2).

[16] Julien Narboux (2007): *Mechanical Theorem Proving in Tarski's Geometry*. In: *Proceedings of Automated Deduction in Geometry 2006*, *LNCS* 4869, Springer.

[17] Loic Pottier Connecting Gröbner Bases Programs with Coq to do Proofs in Algebra, Geometry and Arithmetics. LPAR Workshops. 2008.

[18] Goran Predović (2008): *Automated geometry theorem proving based on Wu's and Buchberger's methods*. *Master thesis*, 4869, Faculty of Mathematics, University of Belgrade.

[19] Dongming Wang (2004): *GEOTHER 1.1: Handling and Proving Geometric Theorems Automatically*. In: *Automated Deduction in Geometry*, *Lecture Notes in Artificial Intelligence* 2930, Springer-Verlag.

[20] Sean Wilson & Jacques Fleuriot (2005): *Combining Dynamic Geometry, Automated Geometry Theorem Proving and Diagrammatic Proofs*. In: *Workshop on User Interfaces for Theorem Provers (UITP)*.

[21] Wen-Tsün Wu (1978): *On the decision problem and the mechanization of theorem proving in elementary geometry*. *Scientia Sinica* 21.

[22] Zheng Ye, Shang-Ching Chou & Xiao-Shan Gao (2008): *An Introduction to Java Geometry Expert*. In: *Automated Deduction in Geometry*.

User Guidance Generated from "Computation plus Deduction" — the Learner's Perspective

Walther Neuper
Institute for Software Technology
University of Technology
Graz, Austria
neuper@ist.tugraz.at

1 Introduction

Herein "learners" are students at high school or at university concerned with applying mathematics to problems in science, and more generally are persons who want to gain insight of how mathematics is working and what mathematics is all about.

"User guidance" addresses software supporting learners in their endeavour: specify a problem (provide in/output items and pre/postcondition more or less interactively) and stepwise construct a result and (more ore less explicitly) justify the steps. Such software either simply addresses a particular problem and handles variants of user input each by specific code; or such software addresses a wide range of mathematics — this poses a significant challenge for software design. Very few systems [5, 1] faced this challenge, and still they are far from perfect.

The two above mentioned systems [5, 1] assemble various concepts and techniques from artificial intelligence and try to mimic human tutors and human teachers. The approach discussed here is quite different: it considers software as a model of mathematics — that means, respective knowledge is represented in software such that it preserves as much of the deductive/algorithmic/applicative structure as possible and such that algorithms are accessible stepwise together with justifications for each step. Then the issue is to design the model such that interactive investigation of the system at work enables learning.

The idea of software as model of mathematics has been discussed for an educational audience in [6]; here the discussion will be from a technical point of view. Thus the question is posed: What are the general concepts in software technology appropriate to represent mathematics ?

The concept of "computation" represents the algorithmic aspect of mathematics. This fact is successfully realised by computer algebra systems (CAS), which implement most of the important algorithms of mathematics nowadays. However, this fact comes hardly to bear in educational systems for two reasons: (1) CAS algorithms strive for generality and thus many of them use deep knowledge; more elementary algorithms frequently lack generality, so this is a design issue. (2) CAS immediately show up with a result, by the way undermining students' motivation for in-depth study; so making intermediate steps accessible is another design issue.

The concept of "deduction" represents the logical aspect of mathematics. Software realising this aspect are computer theorem provers (CTP). Interestingly, automated theorem proving (ATP) techniques are being integrated into dynamic geometry systems (DGS) presently, but not yet in software operating on formulas. In calculations of applied mathematics the deductive aspect is reduced to the question: can this theorem applied to that formula(s) ? However, there is also the issue of checking a step input by a user to a calculation, and we shall see the role of deduction in this case below.

Submitted to:
THedu'11

So the question can be posed at the end of this introduction: How to combine computation and deduction to a general technique for automatic generation of user guidance ?

2 An Interpreter Combining Computation and Deduction

This section briefly introduces the idea how to combine computation and deduction in order to automatically generate user guidance.

"Interpretation" concerns a programming language, which is purely functional and has no in/output statements, but access to simplifiers, equation solvers and other CAS-like functions based in CTP-like logical rigor [4]. A programmer using such a language can focus on mathematics and algorithms, and is not distracted by concerns of interaction due to the lack of in/output statements.

The **computation** of such a program not only creates a result, but also intermediate steps of the respective calculation, and stores the steps in a calculation reflecting the logical dependencies of the steps. Since the steps of a calculation are generated in a well-known way — how does this feature new ways of learning, how comes interaction into play, how can a learner investigate variants of steps by trial and error, at which point of computation can the user input steps ?

Handling interaction is shifted from the program to the interpreter of a program: The interpreter works in a kind of debug mode, stepping from breakpoint to breakpoint (the breakpoints are those statements in the program which create the steps of the calculation). A the breakpoints the steps are presented to the user and control is passed over to her or him.

At a breakpoint the user has free choice over many possibilities to investigate the calculation constructed so far: theorems which justify the steps, theories where the theorems have been proved, etc. And the learner is free to input a step on his or her own, a theorem to be applied or a formula. Such input poses the challenge known from debugger construction: Which input allows to resume execution of the program (i.e. continuing computation of further steps) and which input does not ?

The answer to this question proposed here invokes **deduction**: Checking user input creates a proof situation: Input states a lemma ("invent and verify") which must be derived from the context of the calculation created so far. In order to allow such a proof, rich logical contexts are required. Such contexts are initialised by the preconditions (see problem specification above) and extended with partiality conditions etc at certain steps of computation.

Deduction as implemented in proof-tools of CTPs is, given appropriate logical context, the most general and most powerful technology to check user input in mathematics at the present state-of-the-art.

3 Some Selected Definitions

The concept of an interpreter as introduced above needs rigorous formal description for proper implementation and further research. Since the components of the concept are standard and only the combination produces the specific functionality, such a description starts from the users' (the learners') perspective most appropriately. This will be done in §4 below, but first come the underlying definitions.

When approaching stepwise problem solving in applied mathematics from the learners' perspective it turns out, that a surprising amount of notions (specification, calculation, etc) seems not available in definitions appropriate for this purpose (to our best knowledge). About a dozen of definitions are required; the full number is beyond the limitations of space in this extended abstract. So there is a preliminary selection.

Problems need to be specified in order to enable a software system to automatically solve the problem (and derive user guidance from that).

Definition 1 (Specification)A quintuple $(I, p(I), O, q(I,O), r(I,O))$ is called a specification, where I is a set of *input* values or of variables considered arbitrary but fixed, $p(I)$ is the *precondition*, O a set of *output* variables, $q(I,O)$ the *postcondition* and $r(I,O)$ a set of *properties*. A specification *is_complete* iff
(i) $I \cap O = \emptyset$
(ii) precondition p and postcondition q are well-formed
(iii) $p(I)$ evaluates to true or is indeterminate (but never false).

Educational experience shows that specification is the hardest part of problem solving for many students. The following definition concerns data, which can be provided by an author and used by the system for user guidance or for even hiding specification behind the scenes.

Definition 2 (Problem Declaration) A quadruple (U, thy, pbl, met) is called a declaration of a problem, where the list U of items is called a *formalization* and (thy, pbl, met) is a 3-dimensional pointer into the knowledge base of theories, problemclasses and methods respectively.

Further definitions on problemclass, calculation, method and interpreter can be found in [7].

4 A Rigorous Formalism for Describing Interaction

When stepwise solving a problem in interaction with a math assistant, a learner follows a contract with the assistant. The contract language of Back and von Wright [2] will be used to describe interaction with the system.

Contracts describe the responsibilities of the partners, called angel and demon. Taking the perspective of the learner, the learner is the angel and the system is the demon. Predicate assertions $\{p\}$ and assumptions $[p]$ define conditions the angel, respectively the demon, must satisfy. Relational updates are related to the partners by $\{R\}$ and $[R]$ analogously; for instance, $\{p\};[R]$ denotes pre-postcondition specifications. The contract language is summarised by the following BNF grammar, where p is a predicate and R a relation:

$$C := \{p\} \mid [p] \mid \{R\} \mid [R] \mid C;C \mid C \sqcup C \mid C \sqcap C \mid \mu X \cdot C$$

where μ is the fix-point operator expressing several patterns of interaction. This is an example contract:

$$\{x, e := x', e' \mid x' \geq 0 \wedge e' > 0\}; [x := x' \mid -e \leq x - x'^2 \leq e]$$

This contract requires that the angel first chooses new values for x and e; then the demon is given the task of computing the square-root in the variable x.

Semantics of the contract statements is defined by weakest precondition predicate transformers. A predicate transformer $C : (\Gamma \to Bool) \to (\Gamma \to Bool)$ is a function mapping postcondition predicates to precondition predicates. Given a postcondition q, a precondition p and $f.x$ denoting function application, the following definitions cover the semantics of the constructs used in this paper:

$$\begin{array}{rcll} \{p\}.q & \equiv_{def} & p \cap q & \text{(assertion)} \\ [p].q & \equiv_{def} & \neg p \cup q & \text{(assumption)} \\ (C_1;C_2).q & \equiv_{def} & C_1.((C_2.q) & \text{(sequential composition)} \\ (C_1 \sqcup C_2).q & \equiv_{def} & C_1.q \cup C_2.q & \text{(angelic choice)} \\ (C_1 \sqcap C_2).q & \equiv_{def} & C_1.q \cap C_2.q & \text{(demonic choice)} \end{array}$$

In this semantics, the breaching of a contract by the angel agent means, that the weakest precondition is *false*. If the demon breaches the contract, the weakest precondition is trivially *true*, i.e. the angel has fulfilled his contract anyway.

In order to make the description of interactions short, we shall use two specific variables *in* and *out* denoting input and output respectively as visible to the learner

$$\{x := x' \,|\, x' = in\}; [y := y' \,|\, y' = out]$$

and we even shall shorten the above to

$$\{in := x'\}; [out := y']$$

5 A Glimpse at Interaction

The overall behaviour of a math assistant employing the interpreter introduced above can be described as follows (the reader is advised to read both simultaneously, the verbal description and the subsequent formal description):

(1) **Learn by investigation — and don't intervene with the working system:** The computational part of the system 'knows how to stepwise solve' a selected problem, if an author has prepared a problem declaration $d = (U, thy, pbl, met)$ (Def.2 together with respective knowledge) and if the learner selects the prepared problem by $KNOW.exl = d$. Then the system is obliged to come up with a final step r after repeatedly pushing a button do_next^*. This final r is checked against the postcondition $p(I,O)$, and what happens before in detail is described later.

In this case the learning opportunities arise from the systems' offer to be asked for the knowledge it uses at any step, where *thy* points to deductive knowledge (e.g. a theorem used in rewriting), *pbl* to the current problemclass (embedded into a tree of adjacent problems), *met* to the method which generates the next step — probably generates the step, since the learner is free to switch to case (2) at any step.

(2) **Learn by trial and error — and backtrack when got stuck:** The contracts *Specify* and *Calculate* provide plenty of freedom in interaction for the learner; the more detailed description of the contracts shows, that input during trial and error is checked by the deductive part of the system as generously and liberally as possible at the state of the art in CTP. So, after having (more or less interactively) completed a specification $s = (I, p(I), O, q(I,O), r(I,O))$, the system can use it to check the postcondition and to assure a correct result r.

However, learners' creativity in general goes beyond the capabilities of machines, but there is no 'general mathematical problem solver'. So, user input might well get stuck with an input formula *not_derivable* or with an input tactic which is even applicable, but makes the system *helpless* which means, that the button *do_next* cannot respond with a step.

In this case the learning opportunities arise from trial and error — and if got stuck, arising from backtracking to a state where *do_next* works again.

Comparing the two cases, the most important part of the formula below is the learners freedom (angelic choice $\sqcup$) to switch between the two cases any time:

(1) $\{\ in := exl\ \}$;
$\{\ in := do\_next^*\}$;
$[\ \exists d.\ KNOW.exl = d\ \wedge\ d = (U, thy, pbl, met)\ \wedge$
$\exists l, s, r.\ KNOW.pbl = l\ \wedge\ d\ instantiates\ l = s\ \wedge$
$s = (I, p(I), O, q(I,O), r(I,O))\ \wedge\ r = (\texttt{Check\_Postcond}, f^r)$
$\Rightarrow q(I,O)|_f\ ]$; $[\ out := r\ ]$
$\sqcup$

(2) $Specify$; $Calculate$;
$[\ \exists s, r.\ s = (I, p(I), O, q(I,O), r(I,O))\ \wedge\ r = (\texttt{Check\_Postcond}, f^r)$
$\Rightarrow q(I,O)|_f\ ]$; $[\ out := r\ ]$
$\sqcap$
$[\ out := not\_derivable\ ] \sqcap [\ out := helpless\ ]$

If an author has not prepared knowledge for a problem a learner wants to solve, then the (hard) way through case (2) is the only one — hard, because the system cannot help at the beginning: after input of a theory *thy* only help with syntax and type checking, after input of a specification *s* with searching for a method, etc.

The above formalism allows refinement [3] down to conceptional details for further research and technical details for implementation — both concerns ongoing work.

References

[1] John R. Anderson (2008): *Intelligent Tutoring and High School Mathematics*. Technical Report 20, Carnegie Mellon University, Department of Psychology. Available at `http://repository.cmu.edu/psychology/20`.

[2] Ralph-Johan Back, Anna Mikhailova & Joakim von Wright (1999): *Reasoning about Interactive Systems*. In J. Wing, J. Woodcock & J. Davies, editors: *Proceedings of the World Conference on Formal Methods (FM99)*, *LNCS* 2, Springer-Verlag, Toulouse, France, pp. 1460–1476, doi:10.1007/3-540-48118-4_27.

[3] Ralph-Johan Back & Joakim von Wright (1998): *Refinement Calculus: A Systematic Introduction*. Springer-Verlag. Available at `http://www.amazon.com/Refinement-Calculus-Systematic-Introduction-Computer/dp/0387984178/`. Graduate Texts in Computer Science.

[4] Florian Haftmann, Cezary Kaliszyk & Walther Neuper (2010): *CTP-based programming languages ? Considerations about an experimental design*. *ACM Communications in Computer Algebra* 44(1/2), pp. 27–41, doi:10.1145/1838599.1838621.

[5] E. Melis & J. Siekmann (2004): *ActiveMath: An Intelligent Tutoring System for Mathematics*. In L. Rutkowski, J. Siekmann, R. Tadeusiewicz & L.A. Zadeh, editors: *Seventh International Conference 'Artificial Intelligence and Soft Computing' (ICAISC)*, *LNAI* 3070, Springer-Verlag, pp. 91–101, doi:10.1007/978-3-540-24844-6_12.

[6] Walther Neuper (2010): *Common Grounds for Modelling Mathematics in Educational Software*. *Int. Journal for Technology in Mathematics Education* 17(3). Available at `http://www.ist.tugraz.at/projects/isac/publ/ijtme09.pdf`.

[7] Walther Neuper (2011): *Interaction on Math Assistants based on Lucas-Interpretation — the Learner's Perspective*. Available at `http://www.ist.tugraz.at/projects/isac/publ/lucas-interp-1103.pdf`. Preliminary version.

The GF Mathematics Library

Jordi Saludes
Universitat Politècnica de Catalunya
Sistemes Avançats de Control and MA2
jordi.saludes@upc.edu

Sebastian Xambó
Universitat Politècnica de Catalunya
MA2, Edifici Omega, Barcelona (Spain)
sebastia.xambo@upc.edu

The aim of this paper is to present the *Mathematics Grammar Library*. The main points are the context in which it originated, its current design and functionality and the present development goals. We also comment on possible future applications in the area of artificial mathematics assistants.

1 Introduction

An archetypal meeting point for natural language processing and mathematics education is the realm of *word problems* ([1, 14]), a realm in which mechanised mathematics assistants (MMA) are expected to play an ever more prominent role in the years to come. The Mathematics Grammar Library (MGL) presented in this paper is not an MMA, but our working hypothesis is that it is a good basis on which to build useful MMA's for learning and teaching (cf. [15, 16] for some general clues on e-learning technologies).

In fact, we regard MGL as a technology for providing multilinguality to dialog systems capable of helping students in learning how to solve word problems. This assessment is based on the MGL potential capabilities for dealing effectively with a mixture of text and mathematical expressions, and also for managing interactions with ancillary CAS systems.

To be more specific, the current general aim for MGL is to provide natural language services for mathematical constructs at the level of high school and college freshmen linear algebra and calculus. At the present stage, the concrete goal is to provide rendering of simple mathematical exercises in multiple languages (see [7] for a demo).

2 Origin

For a closer view of MGL, let us look briefly at its origins. The idea behind MGL was born, to a good extend, on reflecting about one of the key results of the WEBALT project (see [5, 3, 9]). In summary, the unfolding of this reflection went as follows.

One of the aims of WEBALT was to produce a proof-of-concept platform for the creation of a multilingual repository of *simple* mathematical problems with guaranteed quality of the (machine) translations, in both linguistic and mathematical terms. The languages envisioned were Catalan, English, Finnish, French, Italian and Spanish. Of these, Finnish, with its great complexities, could not be raised to the same level of functionality as the others.

The WebALT prototype was successful and, as far as we know, that endeavour brought about the first large application of the GF system ([10, 11]) for the multilingual translation of simple mathematical questions. The powerful GF scheme, based on the interlocking of abstract and concrete grammars, was found to be a very sound choice, but the solution had several shortcommings that could not be addressed in that project. For the present purposes, the three that worried us the most are the following:

DRAFT May 5, 2011

- Recent developement of the GF system made our grammars go out of sync.
- The library was monolithic, hampering its usability and maintenance.
- It included too few languages, especially as seen from an Europan perspective.

The springboard for the present library was the need to properly solve these problems, inasmuch as this was regarded as one of the most promissing prerequisites for all further advanced developments in machine processing of mathematical texts. Thus the main tasks were:

- To design a *modular* mathematics library structured according to the semantic standards (content dictionaries) of OPENMATH (see [8] and section 4 below).
- To code it in the much cleaner modern GF release for the few languages mentioned above, and
- To write new code for a few additional languages (Bulgarian, Finnish, German, Romanian and Swedish).

The first two points amount to a tidying of the original WEBALT programming methods. The third point represents not merely an addition of a few more languages, but a thourough testing of the methods and procedures enforced in the preceding steps. This testing is important in order to secure the rules for the inclusion of further languages and for a controlled uniform extension of the available grammars.

At the end, we seek to have a modular and efficient library able to deliver services for a multitude of natural languages to mathematical content providers.

3 Some basic GF notions

For convenience, we include a few notions about the GF system that will ease the considerations about MGL in the next section. For a thourough reference, see [11].

GF is a functional language for multilingual grammar applications where:

- Productions are declared in *abstract modules* whose types are derived from the Martin-Löf type theory;
- Definitions of these abstract productions are given in a *concrete module* specific to each of the targeted languages.

This way, any GF application begins by specifying its abstract syntax. This syntax contains declarations of *categories* (the GF name for types) and *functions* (the GF name for constructor signatures) and has to capture the *semantic structure* of the application domain. For example, to let `Nat` stand for the type of natural numbers and `Prop` for propositions about natural numbers, the GF syntax is

```
cat Nat, Prop ;
```

The Peano inductive construction of the naturals can be expressed as follows:

```
fun
   Zero : Nat ;
   Succ : Nat -> Nat ;
```

The signatures for ‘even number’ and ‘prime number’ can be captured with

```
fun
   Even, Prime : Nat -> Prop ;
```

So we can now express, for example: 'the successor of the successor of zero is prime'.

Finally, we can abstract the logical 'not', 'and' and 'or' as follows:

```
fun
     Not : Prop -> Prop ;
     And, Or : Prop -> Prop -> Prop ;
```

4 The library

MGL is a collection of GF modules which are being developed in the context of the MOLTO project [2]. The categories used in these modules are in correspondence with all possible combinations of **Variable** and **Value** with the mathematical types **Number**, **Set**, **Tensor** and **Function**. Thus the category `VarNum` denotes a numeric variable like x, while `ValSet` denotes an actual set like "the domain of the natural logarithm". The distinction between variables and values allows us to type-check productions like lambda abstractions that require a variable as the first argument. Obviously variables can be promoted to values when needed.

Other categories stand for propositions, geometric constructions and indices.

The library is organised in a matrix-like form, with an horizontal axis ranging over the targeted natural languages. At the moment these are: Bulgarian, Catalan, English, Finnish, French, German, Italian, Romanian, Spanish and Swedish.

The vertical axis is for complexity and contains, from bottom to top, three layers:

1. *Ground*. It deals with literals, indices and variables.
2. *OpenMath*. It is modelled after the OPENMATH *Content Dictionaries* (CD's). Each CD defines a collection of mathematical objects. This is a *de facto* standard for mathematical semantics and for each *Content Dictionary* there is a corresponding module in this layer. The CD's are collected by the *OpenMath consortium* [8].
3. *Operations*. This layer takes care of simple mathematical exercises. These appear in drilling materials and usually begin with directives such as 'Compute', 'Find', 'Prove', 'Give an example of', etc.

The following tree is an example belonging to the *OpenMath* layer:

```
mkProp
(lt_num
  (abs (plus (BaseValNum (Var2Num x) (Var2Num y))))
  (plus (BaseValNum (abs (Var2Num x)) (abs (Var2Num y)))))
```

When linearized with the Spanish concrete grammar, it yields

> El valor absoluto de la suma de x y de y es menor que la suma del valor absoluto de x y del valor absoluto de y

Similarly, the tree

```
DoSelectFromN
  (Var2Num y)
  (domain (inverse tanh))
  (mkProp
```

```
    (gt_num
    (At cosh (Var2Num y))
     pi))
```

gives, when linearized with the English concrete grammar:

> Select y from the domain of the inverse of the hyperbolic tangent such that the hyperbolic cosine of y is greater than pi.

5 Conclusions and further work

After a first step in which the main concern was tidying and modularizing the WEBALT prototype for simple mathematics exercises for the languages Catalan, English, French, Italian and Spanish, we have extended it, in a second step, with the languages Bulgarian, Finnish, German, Romanian and Sweedish. In this note we have described the GF library produced so far for multilingual mathematical text processing, which we call MGL (Mathematics Grammar Library). We have also indicated how it originated in the WEBALT project, its relation to GF, and its present functionality.

Further work has three main lines:

- Addition of new languages, like Danish, Dutch, Norwegian, Polish, Portuguese, Russian, ... This is a continuation of the first two steps referred to above and our assessment is that it can be done reliably with the methods and procedures established so far. To some extent, the library modules for a new language can be generated automatically up to a point from lexical items provided by natives in that language.
- Describing a controlled procedure for the uniform and reliable extension of the grammars according to new semantic needs. This is an important step that is being researched from several angles. One important point is to ascertain when a piece of mathematical text requires functionalities (categories, constructors, operations) not covered yet.
- Advancing in the use of MGL for the production of ever more sophisticated artificial mathematics assistants. This is also the focus of current research that includes a collaboration with statistical machine translation methods, as in principle they can suggest grammatical structures out of a corpus of mathematical sentences.

References

[1] *Word problem (mathematics education)*. Available at `http://en.wikipedia.org/wiki/Word_problem_(mathematics_education)`.

[2] (2011): *Multilingual Online Translation*. Available at `http://www.molto-project.eu`.

[3] O. Caprotti (2006): *WebALT! Deliver Mathematics Everywhere*. In: *Proceedings of SITE 2006*.

[4] O. Caprotti, W. Ng'ang'a & M. Seppälä (2005): *Multilingual technology for teaching mathematics*. In: *Proceedings of the International Conference on Engineering Education, Instructional Technology, Assessment, and E-learning (EIAE 05)*.

[5] O. Caprotti & M. Seppälä (2006): *Multilingual Delivery of Online Tests in Mathematics*. In: *Proceedings of Online Educa Berlin 2006*.

[6] L. Carlson, J. Saludes & A. Strotmann (2005): *State of the art in multilingual and multicultural creation of digital mathematical content*. Available at `http://webalt.math.helsinki.fi/content/e16/e301/e305/Deliverable1.2_eng.pdf`.

[7] Thomas Hallgren & Jordi Saludes (2010): *Math bar online*. Available at `http://www.grammaticalframework.org/demos/minibar/mathbar.html`.

[8] Paul Libbrecht (2010): *OpenMath*. Available at `http://www.openmath.org/`.

[9] W. Ng'ang'a (2006): *Multilingual content development for eLearning in Africa*. In: *eLearning Africa: 1st Pan-African Conference on ICT for Development, Education and Training*.

[10] Aarne Ranta (2010): *Grammatical Framework*. Available at `http://www.grammaticalframework.org/`.

[11] Aarne Ranta (2011): *Grammatical Framework: Programming with Multilingual Grammars*. CSLI Publications, Stanford.

[12] A. Strotmann, W. Ng'ang'a & O. Caprotti (2005): *Multilingual Access to Mathematical Exercise Problems*. In M. Dobreva & J. Engelen, editors: *Electronic Proceedings of the Internet Accessible Mathematical Computation Workshop. ISSAC 2005*, Chinese Academy of Sciences, Bejing, China.

[13] A. Strotmann & M. Seppälä (2005): *Web Advanced Learning Technologies for Multilingual Mathematics Teaching Support*. In M. Dobreva & J. Engelen, editors: *ELPUB2005. From Author to Reader: Challenges for the Digital Content Chain. Proceedings of the 9th ICCC International Conference on Electronic Publishing*, Peeters Publishing, Leuven-Heverlee (Belgium).

[14] L. Verschffel, B. Greer & E. De Corte (2000): *Making sense of word problems*. Taylor & Francis.

[15] S. Xambó, H. Bass, G. Bolaños, R. Seiler & M. Seppälä (2006): *E-Learning Mathematics*. In: *Proceedings of the ICM-2006 (Volume III)*, European Mathematical Society, pp. 1743–1768.

[16] S. Xambó, O. Caprotti & M. Seppälä (2008): *Toward autonomous learners of mathematics*. In J. M. Borwein, E. M. Rocha & J F Rodrigues, editors: *Communicating Mathematics in the Age of Digital Libraries*, A K Peters, pp. 239–252.

WebGeometryLab

Vanda Santos
ESTGV – IPV
Viseu, Portugal
CISUC, University of Coimbra
Coimbra, Portugal
vsantos@estv.ipv.pt

Pedro Quaresma
CISUC, University of Coimbra
Coimbra, Portugal
pedro@mat.uc.pt

1 Extended Abstract

Introduction The area of geometry allows an easy connection between the abstract and the concrete, between the formal description of a particular geometric construction and its concrete representation.

The dynamic geometry systems (DGSs) allow an easy construction of geometric figures build from free objects and constructed objects, using elementary constructions. The free objects can the be manipulated in such a form that the constructed objects are also changed in such a way that the geometric properties of the construction are preserved. This can be used to emphasise the link between the formal nature of geometry and its models.

The DGSs allow also to perform more complex geometric transformations like translation, reflection, rotations they also allow to build and manipulate a variety of figures, visual demonstrations of theorems, graphs, curves, etc. The advantages of the DGSs in a learning environment are multiple, they are easy to use, they stimulate the creativity and the discovery process. There are multiple DGS available: GeoGebra, Cinderella, GeometerSketchpad, C.a.R., Cabri and GCLC [2, 3, 4, 7, 8] to name the most used.

The use of computational methods in the teaching of geometry should be viewed as a new way of conceiving education, i.e., a new tool for assessment/diagnosis during the lessons. In the high-school curriculum in Portugal the use of such tools is praised. Quoting from the official curriculum specification[1] [1].

> The computer, by its own potential, namely in the areas of dynamic geometry, function representation and simulation, allow activities, not only activities of exploration and research but also activities of recovery development, in such a way that it constitute an important asset to teachers and students, its use should be considered obligatory in this curriculum.

The Pythagoras theorem is an example where the DGSs can be used in a very fruitful way. The dynamic component allow us to work the visuals demonstrations of this theorem (see the videos in `www.cs.wichita.edu/~ye`). This theorem allow us to emphasise the links between the formal specifications in geometry and its visualisations.

WebGeometryLab The "Web Geometry Laboratory" (*WebGeometryLab*)[2] is an Web environment, that integrates a DGS program and a database, aiming to provide a learning environment in geometry with individualised memory.

[1] Translated from the Portuguese text.

[2] `http://hilbert.mat.uc.pt/WebGeometryLab/index.php`

Submitted to:
THedu'11

The *WebGeometryLab* system will allow the teacher to create, store and provide to its students a set of geometric constructions, it will allow the student to access to the professor's constructions as well as those kept in a personal "scrapbook", a place where the student will keep his/her own construction; solutions of problems placed by the teacher and/or his/her own exploratory activities. The teacher will have also access to the constructions made, or being made, by the students as a way to be able to help the student during a class or to evaluate the work done after class, or even as a mean to broadcast the work done by a student to the rest of the class.

That system will be easy to install on a school server, or even on the teacher's personal computer. It will require:

- a computer hosting a network server (local or global) able to provide access to the Web pages of the *WebGeometryLab*, and a database management system to provide access to the database that gives the individualised memory to the system. Any Linux/MacOS computer will be able to provide such an environment, it will also be possible to build such an environment on top of a MS-Windows system;
- access to a network (local or global). The school network or even, with the help of an wireless access point, a local/classroom wireless network provided by the teacher;
- computers containing a Web browser connected to the network for teachers' and students' access to the system. Nowadays any computer is equipped with such a program, if not, installing one is always possible.

All this can be built using open source programs so, apart from the hardware this has no other costs, and even that, at least in the Portuguese case, it will not be a problem given the easy access to laptops provided by the "e-escola" ("e-school") program.

The *WebGeometryLab* system is organised in three distinct modules:

- the *management module* that allows the system manager to set whose users will have "teachers status" within the system;
- the *teacher's module*, which allows access (with validation) for teachers. Through this module it will be possible to teachers to make the management of students; it provides also access to the DGS in order to set the constructions that will be accessible to all the students, and finally it will give access, in read mode only, to the "scrapbooks" of students;
- the *student's module* provides access, through a validation process, to the students. This modules allows access, read-only, to the constructions provided by the teachers, and also full access to the student's personal "scrapbook"

The *WebGeometryLab* aims to provide a learning environment for geometry using all the potential of the DGSs, all the easiness of access provided by an Web platform and with an individualised memory provided by a database where all the history of each student is kept providing in this way an adaptable blended learning environment.

From the point of view of the server the system aims to be easy to install, maintain and use, from the point of view of the clients, teachers and students, the only feature that is needed is the access to a java capable[3] Web browser, which is, nowadays, irrelevant given the ubiquity of such type of program.

If installed in the school server the *WebGeometryLab* it is not confined to the classroom, it can be opened to the global network, thus enabling its remote use by teachers and students, extending its use

[3]The *WebGeometryLab* uses the GeoGebra Applet

to a blending learning environment, allowing the teacher to set new challenges to be solved outside the classroom, allowing each individual student to work in his/her own pace.

In technical terms the *WebGeometryLab* system was designed and has been implemented as a multi-platform system, i.e. independent of any computer/operating system platform, based on tools: network server (e.g. Apache), PHP, MySQL in the server side, and a browser, and the languages, JavaScript, AJAX and Java in the client side. All this tools/languages are multi-platform. Also in its implementation it is taken into account the need to adapt to different languages, the *WebGeometryLab* is an i18n/l10n (Internationalisation and localisation) system having the English has its base language an relying in translation to adapt the interface to other languages (e.g. the Portuguese).

Defining a network environment that integrates a DGS program and a database, the *WebGeometryLab* provides more then the simple sum of its components: integrating the DGS fully in a Web-environment it opens the use of the DGS to a collaborative blended learning environment; linking all that with a database allow an individualisation of the learning environment with the creation of individualised learning path.

Extending *WebGeometryLab* to a LMS As described above the *WebGeometryLab* will provide an adaptative, collaborative, Web learning environment to Geometry. In spite of the DGSs outstanding features, they do not create a learning environment by themselves, the *WebGeometryLab* goes a step further on that incorporating the DGS and a database in a Web environment implementing some features of a learning environment, nevertheless some important features of a learning environment are still lacking. The learning management systems (LMS) are the proper setting enabling the management of courses, teaching assistance, management and distribution of content to students, in an Web environment with synchronous and asynchronous features.

In the next section we will try to expand on how and why to incorporate *WebGeometryLab* in a LMS.

Learning Managements Systems With the construction of *WebGeometryLab* we wanted to build an Web environment, integrating a DGS and a database, where both teacher and students, have the tools necessary to study Euclidean geometry its theories and models and to be able to understand the bridges connecting these two perspectives, exploring the axiomatic nature of geometry. A learning environment that enables students to test and enhance their knowledge.

The LMSs are courses management systems capable of handling a large number of courses and users. They aim to support the learning process of each student, without physical barrier and asynchronously. They are collaborative learning environments promoting the learning through collaborative efforts.

To merge *WebGeometryLab* in a LMS is the ultimate goal of the work that is being developed. Adding a dynamic geometry tool to this type of system will make the system more rewarding because it will allow students to explore themes of geometry in order to overcome some (not necessarily all) difficulties or shortcomings that may have. Merging the dynamic geometry tool and the database, for an individualised memory, into a LMS will provide teachers and students with an adaptive learning environment for geometry.

Future Work The *WebGeometryLab* is a "work on progress" project, as a first task we need to complete a first prototype of a standalone system capable of being distribute to schools and/or teachers, install and be used by them, such a system should already include a set of geometric constructions a course syllabus to help teachers to organise there work. After that the integration of this system in a learning management system with some samples courses already in placed would be the second task. The integration of

automated theorem proving (ATP) and/or interactive theorem proving (ITP) it is also a wanted feature and it is planned as a task to be pursued in parallel with the described tasks [5, 6].

The goal of this project it is to build a dynamic adaptative learning environment, an environment where the student has at his/her disposal the necessary tools for the study of theories and models of geometry, to understand the differences and connections between these two perspectives, improve their knowledge. A system in which the student can also being challenged by new problems, giving the student the opportunity to develop and improve their study in the area of geometry anywhere, anytime and at their own pace.

References

[1] Departamento do Ensino Secundário (2011): *Matemática A, Cursos Gerais de Ciências Naturais, Ciências e Tecnologias, Ciências Sócio-Económicas*. Technical Report, Ministério da Educação, República Portuguesa.

[2] René Grothmann (2011): *About C.a.R.* `http://compute.ku-eichstaett.de/MGF/wikis/caruser/doku.php?id=history`.

[3] M Hohenwarter (2002): *GeoGebra - a software system for dynamic geometry and algebra in the plane*. Master's thesis, University of Salzburg, Austria.

[4] N Jackiw (2001): *The Geometer's Sketchpad v4.0*. Key Curriculum Press.

[5] Predrag Janičić & Pedro Quaresma (2007): *Automatic Verification of Regular Constructions in Dynamic Geometry Systems*. In: *Automated Deduction in Geometry*, *LNAI* 4869, Springer, Berlin / Heidelberg, pp. 39–51. 6th International Workshop, ADG 2006, Pontevedra, Spain, August 31-September 2, 2006. Revised Papers.

[6] P. Janičić, J. Narboux & P. Quaresma (2011): *The Area Method: a Recapitulation*. *Journal of Automated Reasoning* (to appear), doi:10.1007/s10817-010-9209-7. DOI: 10.1007/s10817-010-9209-7.

[7] Predrag Janičić (2006): *GCLC — A Tool for Constructive Euclidean Geometry and More Than That*. *Lecture Notes in Computer Science* 4151, pp. 58–73, doi:10.1007/11832225_6.

[8] Jürgen Richter-Gebert & Ulrich Kortenkamp (1999): *The Interactive Geometry Software Cinderella*. Springer.

Computer-Assisted Program Reasoning Based on a Relational Semantics of Programs*

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

Extended Abstract

Most systems for program reasoning are based on calculi such as the Hoare Calculus or Dynamic Logic [1] where we generate from a program specification and a program implementation (which is annotated with additional meta-information such as loop invariants and termination terms) those conditions whose verification implies that the implementation indeed meets the specification. The problem is that by such an approach we gain little insight into the program before respectively independently of the verification process. In particular, if the verification attempt is a priori doomed to fail because of errors, inconsistencies, or weaknesses in the program's specification, implementation, or meta-information (which is initially the case in virtually all verification attempts), we will learn so only by unsuccessfully struggling with the verification until some mental "click" occurs. This click occurs frequently very late, because, in the heat of the struggle, it is usually hard to see whether the inability to perform a correctness proof is due to an inadequate proving strategy or due to errors or inconsistencies in the program. Actually, it is usually the second factor that contributes most to the time spent and frustration experienced; once we get the specification/implementation/meta-information correct, the verification is a comparatively small problem. We have frequently observed this fact in our own verification attempts as well as in those performed by students of computer science and mathematics in courses on formal methods.

We therefore advocate an alternative approach where we insert between a program and its verification conditions an additional layer, the denotation of the program [4] expressed in a declarative form. The program (annotated with its meta-information) is translated into its denotation from which subsequently the verification conditions are generated. However, even before (and independently of) any verification attempt, one may investigate the denotation itself to get insight into the "semantic essence" of the program, in particular to see whether the denotation indeed gives reason to believe that the program has the expected behavior. Errors in the program and in the meta-information may thus be detected and fixed prior to actually performing the formal verification.

More concretely, following the relational approach to program semantics [2], we model the effect of a program (command) c as a binary relation $[\![c]\!]$ on program states which describes the possible pairs of pre- and post-states of c. Such a relation can be also described in a declarative form by a logic formula f_r with denotation $[\![f_r]\!]$. Thus a formal calculus is devised to derive from a program c a judgment $c : f_r$ such that $[\![c]\!] \subseteq [\![f_r]\!]$. For instance, we can derive

$$\texttt{x=x+1} : \mathsf{var}\ x = \mathsf{old}\ x + 1$$

*Sponsored by the Austrian Science Fund (FWF) in the frame of the DK "Computational Mathematics" (W1214).

Submitted to:
THedu'11

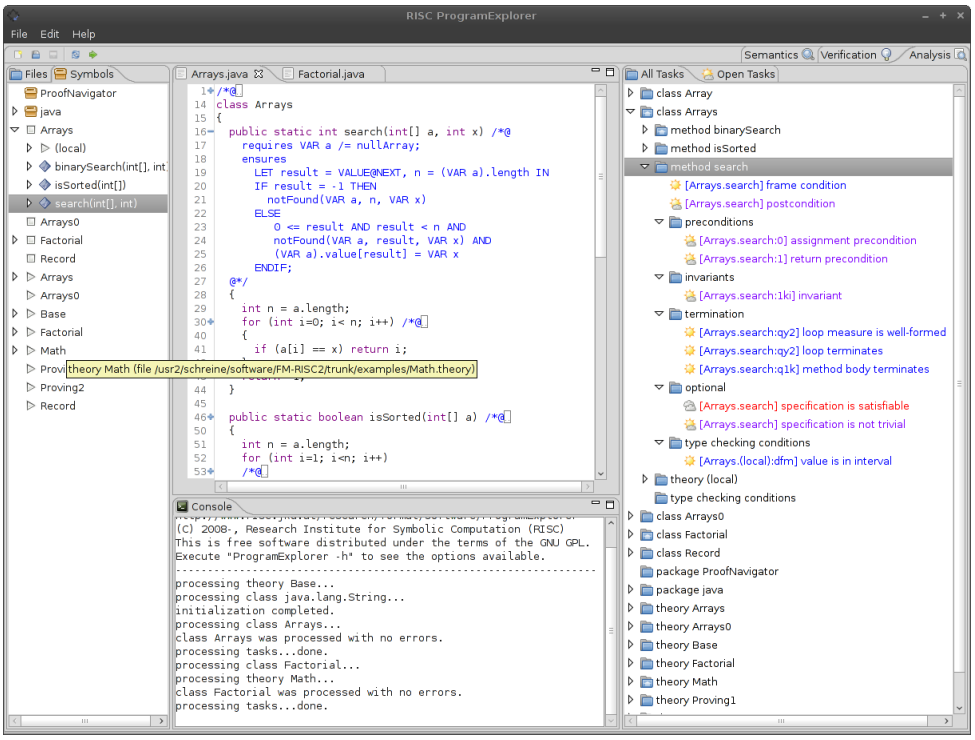


Figure 1: The RISC ProgramExplorer

where the logic variable old x refers to the value of the program variable x in the prestate of the command and the logic variable var x refers to its value in the poststate. In this way, we can constrain the allowed state transitions, i.e. handle the partial correctness of programs. To capture also total correctness, we introduce the set of states $\langle\!\langle c \rangle\!\rangle$ on which the execution of c must terminate ($\langle\!\langle c \rangle\!\rangle$ is a subset of the domain of $[\![c]\!]$). Such a set can be also described in a declarative form by a logic formula (a state condition) f_c. Thus we derive a judgment $c \downarrow f_c$ such that $[\![f_c]\!] \subseteq \langle\!\langle c \rangle\!\rangle$. In this fashion, the pair of formulas f_r and f_c captures the semantic essence of c in a declarative form that is open for inspection and manipulation.

We have implemented this idea in a comprehensive form in the *RISC ProgramExplorer*[1], a new program reasoning environment for educational purposes which encompasses the previously developed *RISC ProofNavigator* as an interactive proving assistant [5]. The RISC ProgramExplorer supports reasoning about programs written in a restricted form of Java (including support for control flow interruptions such as `continue`, `break`, `return`, and `throw`, static and dynamic methods, classes and a restricted form of objects) and specified in the formula language of the RISC ProofNavigator (which is derived from PVS [3]). The system is currently in beta state, a first release under the GNU Public License will be available by July 2011 and will be subsequently used in regular courses. A screenshot of the software is given in Figure 1; in the remainder of this abstract, we will first sketch the formalism on which the RISC ProgramExplorer is based and then give an illustrative example.

For the purpose of this presentation, we use a simple command language without control flow interruptions and method calls; a command c can be formed according to the grammar

$$c ::= x = e \mid \{\texttt{var}\ x;\ c\} \mid \{c_1\,;c_2\} \mid \texttt{if}\ (e)\ \texttt{then}\ c \mid \texttt{if}\ (e)\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{while}\ (e)^{f,t}\ c$$

[1] `http://www.risc.jku.at/research/formal/software/ProgramExplorer`

$$\frac{c : [f]^{xs}_{g,h} \qquad x \notin xs}{c : [f \wedge \mathsf{var}\ x = \mathsf{old}\ x]^{xs \cup \{x\}}_{g,h}} \qquad \frac{e \simeq_h t}{x = e : [\mathsf{var}\ x = t]^{\{x\}}_{\mathsf{true},h}} \qquad \frac{c : [f]^{xs}_{g,h}}{\texttt{\{var } x\texttt{; } c\texttt{\}} : [\exists x : f]^{xs \setminus x}_{g, \forall x : h[x/\mathsf{old}\ x]}}$$

$$\frac{c_1 : [f_1]^{xs}_{g_1,h_1} \qquad c_2 : [f_2]^{xs}_{g_2,h_2} \qquad \mathrm{PRE}(c_2,h_2) = h_3}{\texttt{\{}c_1\texttt{;}c_2\texttt{\}} : [\exists ys : f_1[ys/\mathsf{var}\ xs] \wedge f_2[ys/\mathsf{old}\ xs]]^{xs}_{g_1 \wedge g_2, h_1 \wedge h_3}}$$

$$\frac{e \simeq_h f_e \qquad c_1 : [f_1]^{xs}_{g_1,h_1}}{\texttt{if (}e\texttt{) then } c : [\mathsf{if}\ f_e\ \mathsf{then}\ f_1\ \mathsf{else}\ \mathsf{var}\ xs = \mathsf{old}\ xs]^{xs}_{g_1, h \wedge (f_e \Rightarrow h_1)}}$$

$$\frac{e \simeq_h f_e \qquad c_1 : [f_1]^{xs}_{g_1,h_1} \qquad c_2 : [f_2]^{xs}_{g_2,h_2}}{\texttt{if (}e\texttt{) then } c_1 \texttt{ else } c_2 : [\mathsf{if}\ f_e\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2]^{xs}_{g_1 \wedge g_2, h \wedge \mathsf{if}\ f_e\ \mathsf{then}\ h_1\ \mathsf{else}\ h_2}}$$

$$\frac{\begin{array}{c} e \simeq_h f_e \qquad c : [f_c]^{xs}_{g_c,h_c} \\ g \equiv \forall xs, ys, zs : f[xs/\mathsf{old}\ xs, ys/\mathsf{var}\ xs] \wedge f_e[ys/\mathsf{old}\ xs] \wedge f_c[ys/\mathsf{old}\ xs, zs/\mathsf{var}\ xs] \Rightarrow \\ h[ys/\mathsf{old}\ xs] \wedge f[xs/\mathsf{old}\ xs, zs/\mathsf{var}\ xs] \end{array}}{\texttt{while (}e\texttt{)}^{f,t}\ c : [f \wedge \neg f_e[\mathsf{var}\ xs/\mathsf{old}\ xs]]^{xs}_{g_c \wedge g, h \wedge f[\mathsf{old}\ xs/\mathsf{var}\ xs]}}$$

Figure 2: The Transition Rules

$$x = e \downarrow_{\mathsf{true}} \mathsf{true} \qquad \frac{c \downarrow_g f}{\texttt{\{var } x\texttt{; } c\texttt{\}} \downarrow_g \forall x : f} \qquad \frac{c_1 \downarrow_{g_1} f_1 \quad c_2 \downarrow_{g_2} f_2 \quad \mathrm{PRE}(c_2,f_2) = f_3}{\texttt{\{}c_1\texttt{;}c_2\texttt{\}} \downarrow_{g_1 \wedge g_2} f_1 \wedge f_3}$$

$$\frac{e \simeq_h f_e \qquad c \downarrow_g f}{\texttt{if (}e\texttt{) then } c \downarrow_g f_e \Rightarrow f} \qquad \frac{e \simeq_h f_e \qquad c_1 \downarrow_{g_1} f_1 \qquad c_2 \downarrow_{g_2} f_2}{\texttt{if (}e\texttt{) then } c_1 \texttt{ else } c_2 \downarrow_{g_1 \wedge g_2} \mathsf{if}\ f_e\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2}$$

$$\frac{\begin{array}{c} e \simeq_h f_e \qquad c : [f_c]^{xs}_{g_c,h_c} \qquad c \downarrow_{g_t} f_t \\ g \equiv \forall xs, ys, zs : f[xs/\mathsf{old}\ xs, ys/\mathsf{var}\ xs] \wedge f_e[ys/\mathsf{old}\ xs] \wedge f_c[ys/\mathsf{old}\ xs, zs/\mathsf{var}\ xs] \Rightarrow \\ g_t[ys/\mathsf{old}\ xs] \wedge f_t[ys/\mathsf{old}\ xs] \wedge \mathsf{let}\ n = t[zs/\mathsf{old}\ xs]\ \mathsf{in}\ n \in \mathbb{N} \wedge n < t[ys/\mathsf{old}\ xs] \end{array}}{\texttt{while (}e\texttt{)}^{f,t}\ c \downarrow_g t \in \mathbb{N}}$$

Figure 3: The Termination Rules

where x denotes a program variable, e denotes a program expression, and a while loop is annotated by an invariant formula f and termination term t. As shown in Figures 2, 3, and 4 (where the terms old xs and var xs refer to the sets of values of the program variables xs in the pre-/post-state), we can derive for these commands the following kinds of judgments:

- $c : [f_r]^{xs}_{g,h}$ denotes the derivation of a state relation f_r from command c together with the set of program variables xs that may be modified by c. The derived relation is correct if the derived state-independent condition g holds, and if the derived state condition h holds on the pre-state of c. The rationale for g is is to capture state-independent conditions such as the correctness of loop invariants; the purpose of h is to capture statement preconditions that prevent e.g. arithmetic overflows. These side conditions have to be proved; they are separated from the transition relation f_r to make the core of the relation better understandable.
- $c \downarrow_{g_c} f_c$ denotes the derivation of a state condition (termination condition) f_c from c; the derived condition is correct, if the state-independent condition g_c holds. The purpose of this side condition

$$\frac{c : [f]^{xs}_{g,h}}{\text{PRE}(c, f_q) = \forall xs : f[xs/\mathsf{var}\ xs] \Rightarrow f_q[xs/\mathsf{old}\ xs]}$$

$$\frac{c : [f]^{xs}_{g,h}}{\text{POST}(c, f_p) = \exists xs : f_p[xs/\mathsf{old}\ xs] \wedge f[xs/\mathsf{old}\ xs, \mathsf{old}\ xs/\mathsf{var}\ xs]}$$

Figure 4: The Pre-/Postcondition Rules

is to capture that the value of a loop's termination term does not decrease forever.

- $\text{PRE}(c, f_q) = f_p$ and $\text{POST}(c, f_p) = f_q$ denote derivations that compute from a command c and a condition f_q on the post-state of c a corresponding condition f_p on the pre-state, respectively from c and pre-condition f_p the post-condition f_q. The corresponding rules in Figure 4 show that these conditions can be computed directly from the transition relation of c.

The derivations make use of additional judgments $e \simeq_{f_e} f$ and $e \simeq_{f_e} t$ which translate a boolean-valued program expression e into a logic formula f and an expression e of any other type into a term t, provided that the state in which e is evaluated satisfies the condition f_e (the rules for these judgments are omitted). Formally, the derivations satisfy the following soundness constraints.

Theorem 1 (Soundness) *For all $c \in Command, f_r, f_c, f_p, f_q, g, h \in Formula, xs \in \mathbb{P}(Variable)$, the following statements hold:*

1. *If we can derive the judgment $c : [f_r]^{xs}_{g,h}$, then we have for all $s, s' \in Store$*

$$[\![g]\!] \wedge [\![h]\!](s) \Rightarrow ([\![c]\!](s, s') \Rightarrow [\![f_r]\!](s, s') \wedge \forall x \in Variable \backslash xs : [\![x]\!](s) = [\![x]\!](s')).$$

2. *If we can (in addition to $c : [f_r]^{xs}_{g,h}$) derive the judgment $c \downarrow_{g_c} f_c$, then we have for all $s \in Store$*

$$[\![g]\!] \wedge [\![g_c]\!] \wedge [\![h]\!](s) \Rightarrow ([\![f_c]\!](s) \Rightarrow \langle\!\langle c \rangle\!\rangle(s)).$$

3. *If we can (in addition to $c : [f_r]^{xs}_{g,h}$) also derive the judgment $\text{PRE}(c, f_q) = f_p$ or the judgment $\text{POST}(c, f_p) = f_q$, then we have for all $s, s' \in Store$*

$$[\![g]\!] \wedge [\![h]\!](s) \Rightarrow ([\![f_p]\!](s) \wedge [\![f_r]\!](s, s') \Rightarrow [\![f_q]\!](s')).$$

The semantics $[\![f]\!](s, s')$ of a transition relation f is determined over a pair of states s, s' (and a logic environment, which is omitted for clarity); the semantics of state condition g is defined as $[\![g]\!](s) \Leftrightarrow \forall s' : [\![g]\!](s, s')$ and the semantics of a state independent-condition h is defined as $[\![h]\!] \Leftrightarrow \forall s, s' : [\![h]\!](s, s')$.

As an example, take the following method *fac* computing the factorial of a natural number n (the specification term `VALUE@NEXT` denotes the return value of *fac*):

```
public static int fac(int n) /*@
  requires OLD n >= 0 AND factorial(OLD n) <= Base.MAX_INT;
  ensures VALUE@NEXT = factorial(OLD n); @*/ {
  int i=1; int p=1;
  while (i <= n) /*@
    invariant OLD n >= 0 AND factorial(OLD n) <= Base.MAX_INT
          AND 1 <= VAR i AND VAR i <= OLD n+1 AND VAR p = factorial(VAR i-1);
    decreases OLD n - OLD i + 1; @*/ {
    p = p*i; i = i+1;
  }
  return p;
}
```

Based on the calculus above, the RISC Program Explorer translates the while loop to the following formulas (and also generates tasks for the verification of the various side conditions):

Effects

executes: true, **continues:** false, **breaks:** false, **returns:** false
variables: i, p; **exceptions:**-

Transition Relation

$$\text{var } i = (\text{old } n+1) \wedge \text{old } n \geq 0 \wedge \text{factorial}(\text{old } n) \leq \text{Base.MAX}_{\text{INT}} \wedge 1 \leq \text{var } i$$
$$\wedge$$
$$\text{var } p = \text{factorial}(\text{var } i - 1)$$

Termination Condition

$$(\text{old } n - \text{old } i + 1) \geq 0$$

Here the core of the transition relation is the formula $\mathsf{var}\ i = \mathsf{old}\ n+1 \wedge \mathsf{var}\ p = \mathit{factorial}(\mathsf{var}\ i-1)$ while the termination termination condition is $\mathsf{old}\ n - \mathsf{old}\ i + 1 \geq 0$. Based on this translation, the body of the method *fac* is translated to

Here the core of the transition relation is $\exists i : i = \mathsf{old}\ n+1 \wedge \mathit{value}@\mathsf{next} = \mathit{factorial}(i-1)$ which can be further simplified to $f_r :\Leftrightarrow \mathit{value}@\mathsf{next} = \mathit{factorial}(\mathsf{old}\ n)$; the termination condition can be further simplified to $f_c :\Leftrightarrow \mathsf{old}\ n \geq 0$ (work is going on to improve the automatic simplification). Both f_r and f_c represent the semantic essence of *fac* from which the correctness of the method according to its specification is quite self-evident even before the formal proof is started. More realistic examples seem to indicate that from the construction and simplification of the semantic essence also the further verifications become substantially clearer and perhaps even technically simpler.

References

[1] Bernhard Beckert, Reiner Hähnle & Peter H. Schmitt, editors (2007): *Verification of Object-Oriented Software: The KeY Approach*. Lecture Notes in Computer Science 4334, Springer-Verlag. `http://www.springer.com/computer/ai/book/978-3-540-68977-5`.

[2] Leslie Lamport (2002): *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley. `http://research.microsoft.com/users/lamport/tla/book.html`.

[3] S. Owre, J. M. Rushby & N. Shankar (1992): *PVS: A Prototype Verification System*. In Deepak Kapur, editor: *11th International Conference on Automated Deduction (CADE)*, Lecture Notes in Artificial Intelligence 607, Springer, Saratoga, NY, June 14–18, pp. 748–752. `http://www.csl.sri.com/papers/cade92-pvs`.

[4] David A. Schmidt (1986): *Denotational Semantics – A Methodology for Language Development*. Allyn and Bacon, Boston, MA. `http://people.cis.ksu.edu/~schmidt/text/densem.html`.

[5] Wolfgang Schreiner (2008): *The RISC ProofNavigator: A Proving Assistant for Program Verification in the Classroom*. Formal Aspects of Computing doi:10.1007/s00165-008-0069-4.

Isabelle/PIDE as Platform for Educational Tools

Makarius Wenzel and Burkhart Wolff

Université Paris-Sud 11, LRI, Orsay, France

1 Introduction

Isabelle [9] is an interactive theorem prover platform in the tradition of LCF [3]. Other notable descendants of LCF are HOL4 and Coq, see also [12]. Even after several decades, current systems share the following main traits of the original LCF approach.

1. **Strong logical foundations:** Some well-understood logical basis is taken as starting point, and mathematical theories are explicitly constructed by reduction to first principles. This follows the tradition of "honest toil" in the sense of Bertrand Russel: results are not just postulated as axioms, but derived from definitions as actual theorems.
2. **Free programmability and extensibility:** Derived proof tools can be implemented on top of the logical core, while fully retaining its integrity. This works by the strong type-safety properties of the ML implementation platform of the prover.
3. **Primitive read-eval-print loop:** User interaction works by issuing individual commands, which the prover interprets on the spot and prints results accordingly. Prover commands update an implicit state, which most newer provers allow to *undo* in a linear fashion, to support the well-known proof scripting mode of Proof General [1].

On the one hand, this general architecture proved quite successful in building reasonably large libraries of formal theories (such as the Archive of Formal Proof `http://afp.sf.net` for Isabelle). On the other hand, integrating an LCF-style prover into into a combined *mathematical assistant*, especially one intended for *math education*, poses some challenges.

1. **Interaction:** How can casual users interact with the prover, without getting exposed to the full details of logic implemented on the computer? How can specific interaction scenarios that are relevant for computer-assisted math education be supported, hiding the fact that there is a fully-featured prover engine at the bottom?
2. **Integration:** How can other systems connect to the prover engine? How can we overcome the traditional plumbing of the read-eval-print loop via pipes, typically with synchronous / sequential protocols. How can we proceed to the next generation of integrated mathematical assistants, with sophisticated front-ends and back-ends, using asynchronous / parallel evaluation?

We shall provide an overview of some aspects of recent Isabelle versions that aim to overcome these problems, and facilitate future implementations of mathematical assistants and applications to math education. Thus we address the above weak spot of "primitive read-eval-print loop" in particular, while retaining the inherent strengths of the LCF prover architecture with its strong logical foundations.

One could argue if theorem prover technology should get involved in educational tools at all. For example, major teaching tools for dynamic geometry like Geogebra `http://www.geogebra.org` have

Submitted to:
THedu 2011

managed to do without for many years already. We do not attempt any finals answers to this fundamental question here, but strive to overcome the accidental technical restrictions of traditional provers that made it difficult or impossible to start serious investigations in the first place. We are confident that a logical engine that is easy to integrate into other systems will find its users. In fact, our work is motivated by by previous discussions with tool builders about this very topic.

Subsequently we outline the technological state of the Isabelle platform, and indicate some application scenarios of educational tool development. Many building blocks that are already available in Isabelle2011 were motivated by the *PIDE* project, which aims at a sophisticated *Prover IDE* for LCF-style provers. This can be taken as starting point for further applications.

2 Isabelle/Isar as generic Mathematical Notepad

One of the specific strengths of the Isabelle platform is the Isar proof language [11] that allows to express formal reasoning in a way that is both human-readable and machine-checkable. Unlike other systems in this category (notably Mizar [12]), the Isar proof language is merely an application of a more general framework for structured logical environments: the notion of *Isar proof context* provides flexible means to operate within locally fixed parameters and assumptions. Over the years the general framework has already been re-used for structured specifications and module systems in Isabelle (such as locales and type-classes).

Beyond the default Isar proof language, it is possible to implement domain-specific languages for structured reasoning. To hint at the potential, we show outlines for calculational reasoning and induction in Isabelle/Isar:

```
notepad
begin
  have a = b sorry
  also
  have ... = c sorry
  also
  have ... = d sorry
  finally
  have a = d .
end
```

```
notepad
begin
  fix n :: nat have P n
  proof (induct n)
    case 0 show P 0 sorry
  next
    case (Suc n)
    from ⟨P n⟩ show P (Suc n) sorry
  qed
end
```

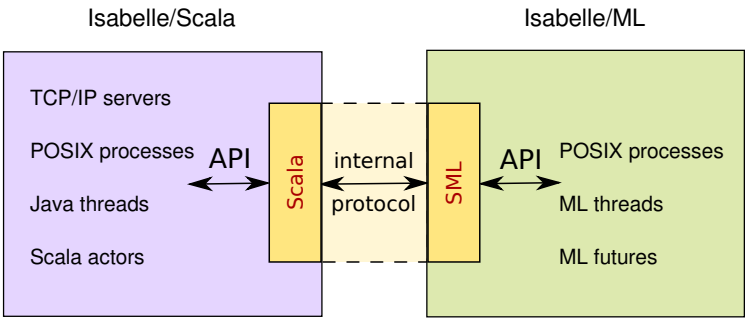
Here the language elements **also** and **finally** for calculational sequences, and the specific *induct* proof method with its symbolic cases are defined in user-space, although they happen to be part of the standard library of Isabelle. Understanding Isabelle/Isar as generic framework for domain-specific formal languages, rather than a particular proof language, tool builders can start to implement their own language elements. The *Isabelle/Isar Implementation* and the *Isabelle/Isar Reference Manual* that are part of the official distribution provide further information.

3 Isabelle/ML versus Isabelle/Scala

Isabelle/ML is both the implementation and extension language of Isabelle/Isar. ML is embedded into the formal Isar context, such that user code can refer to formal entities in the text, to achieve some static type-checking of logical syntax. It is also possible to make ML tools depend on logical parameters and assumptions, and apply them later in a different context with concrete values and theorems.

The relation of Isabelle/ML versus the Isabelle logical framework is best understood as an elaboration of the original LCF approach, where ML was introduced as meta-language to manipulate logical entities with full access to formal syntax.

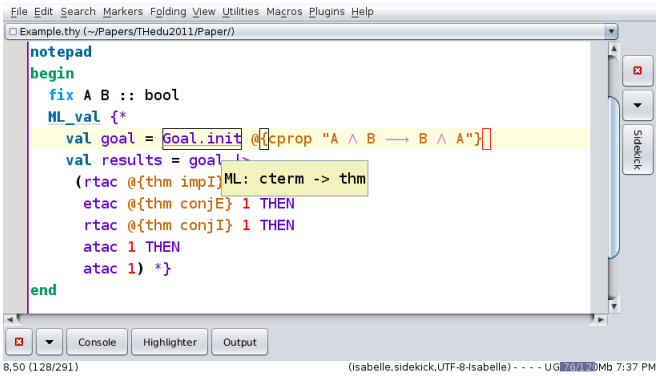
Isabelle/Scala is another layer around the Isabelle prover that was introduced more recently to facilitate *system interaction and integration*. The idea is to wrap the traditional ML prover process into some library for Scala/JVM [6]. Thus the prover can be accessed via some statically-typed Scala API, while the connection between the two different processes (ML back-end versus JVM front-end) is hidden in some *internal protocol* (in contrast to the former attempt of public PGIP [2]). The implementation leverages the existing concurrent and parallel infrastructure of both ML [5] and Scala [4].



The Isabelle/Scala interaction model is *asynchronous*: the front-end can issue batches of document changes to the prover, which will be processed in pipelined and parallel mode. Results are accumulated in monotonic fashion in a persistent document model, which consists of a huge XML tree that is incrementally extended for each document version, and can be visualized on demand according to physical GUI events.

Our mixed ML/Scala architecture allows to re-use decades of research into core prover technology, while integrating with more mainstream front-ends. A typical example is the Prover IDE as implemented in Isabelle/jEdit [10]. Here the existing editor framework jEdit `http://www.jedit.org`, which is based on conventional Java/Swing together with some simple plugin architecture, is connected to the semantic document content provided by the Isabelle process in the background. Thus the user gets an impression of continuous proof checking, with IDE-style visualization of error messages, proof states, results etc. Additional semantic information is attached to the source text, and can be displayed as tooltips, popups, hyperlinks etc. within the editor.

Isabelle/jEdit can be understood as reference application for other projects, and at the same time it can already be used as IDE for Isabelle theory development, which includes embedded Isabelle/ML for add-ons tools. The following screenshot illustrates the Isabelle/Isar/ML IDE aspect.



Here the underlying Poly/ML compiler `http://www.polyml.org` is integrated into the document model of Isabelle/Scala, to provide immediate interactive feedback on the JVM-side. We have designed the principles of rich document annotations to make it easy to upgrade existing back-ends (like Poly/ML or the Isabelle prover core), and avoid re-implementing the same on the JVM-side.

4 Some Application Scenarios

The ongoing development of the Isabelle/Scala layer and the Isabelle/jEdit application demonstrate that it is feasible to build combined mathematical assistants with LCF-style formal basis. We can anticipate the following further application scenarios, which are particularly relevant for educational tool development.

1. Document preparation with rich semantic content, based on the logical context of a given theory library. The idea is to render the markup information that the prover provides for formal source text into XHTML/CSS, or similar technologies. This allows to browse annotated text within a JVM-based HTML browser, for example.
2. Client-side mathematical editors with specific support for typical patterns like calculational reasoning and induction. Such mathematical worksheets should also target at proper mathematical notation, instead of the direct source presentation of the existing Prover IDE. This requires some means for high-quality typesetting of mathematical formulae on the JVM, which are unfortunately not easily available in open-source, though.
3. Rich-client IDE components as extensions to the Prover IDE. The design of jEdit makes it easy to assemble JVM/Swing components as plugins. Thus existing tools like Geogebra that happen to work with the same platform can be easily integrated, at least at the GUI level. The actual logical connection to some theory of geometry is a different issue, e.g. see [7] for some recent work with Coq.
4. Server-side applications using JVM-based web frameworks. Java/Swing GUI components for rich clients are adequate, but the true strength of the JVM platform are its server components. We have only started to investigate the possibilities: JVM-based web servers like Apache Tomcat or Jetty are commonplace; sophisticated application frameworks like Lift `http://liftweb.net` are built on top, and allow easy integration of Scala components in particular.

The web application scenario might turn out particularly interesting for educational purposes, especially in combination with the aspects of document preparation and mathematical editors. For example, Isabelle/Scala could be used as a basis for some mathematical Wiki [8] that can be used by students immediately within their usual browser. Powerful client components, like our existing Prover IDE, also fit into this scenario as high-end authoring tools to produce formal content in the first place.

We encourage implementors of other mathematical tools to develop further ideas, and populate the emerging eco-system around the Isabelle/Isar/ML/Scala/PIDE platform.

References

[1] David Aspinall (2000): *Proof General: A Generic Tool for Proof Development*. In Susanne Graf & Michael Schwartzbach, editors: *European Joint Conferences on Theory and Practice of Software (ETAPS)*, *LNCS* 1785, Springer.

[2] David Aspinall, Christoph Lüth & Daniel Winterstein (2007): *A Framework for Interactive Proof*. In M. Kauers, Manfred Kerber, Robert Miner & Wolfgang Windsteiger, editors: *Towards Mechanized Mathematical Assistants (CALCULEMUS and MKM 2007)*, *LNAI* 4573, Springer.

[3] M. J. C. Gordon, R. Milner & C. P. Wadsworth (1979): *Edinburgh LCF: A Mechanized Logic of Computation*. *LNCS* 78, Springer.

[4] P. Haller & M. Odersky (2006): *Event-Based Programming without Inversion of Control*. In: *Joint Modular Languages Conference*, Springer LNCS.

[5] David C. J. Matthews & Makarius Wenzel (2010): *Efficient Parallel Programming in Poly/ML and Isabelle/ML*. In: *ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming (DAMP 2010), co-located with POPL*, ACM Press.

[6] M. Odersky et al. (2004): *An Overview of the Scala Programming Language*. Technical Report IC/2004/64, EPF Lausanne.

[7] Tuan Minh Pham & Yves Bertot (2010): *A combination of a dynamic geometry software with a proof assistant for interactive formal proofs*. In C. Sacerdoti Coen & D. Aspinall, editors: *User Interfaces for Theorem Provers (UITP 2010)*, ENTCS. FLOC 2010 Satellite Workshop.

[8] J. Urban, J. Alama, P. Rudnicki & H. Geuvers (2010): *A Wiki for Mizar: Motivation, Considerations, and Initial Prototype*. In: *Conference on Intelligent Computer Mathematics (CICM 2010)*, *LNCS* 6167, Springer.

[9] M. Wenzel, L.C. Paulson & T. Nipkow (2008): *The Isabelle Framework*. In: *Theorem Proving in Higher Order Logics (TPHOLs 2008)*, LNCS, Springer. Available at `http://www4.in.tum.de/~wenzelm/papers/isabelle-overview.pdf`. Invited paper.

[10] Makarius Wenzel (2010): *Asynchronous Proof Processing with Isabelle/Scala and Isabelle/jEdit*. In C. Sacerdoti Coen & D. Aspinall, editors: *User Interfaces for Theorem Provers (UITP 2010)*, ENTCS. FLOC 2010 Satellite Workshop.

[11] Markus Wenzel (1999): *Isar — a Generic Interpretative Approach to Readable Formal Proof Documents*. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin & L. Thery, editors: *Theorem Proving in Higher Order Logics (TPHOLs 1999)*, *LNCS* 1690, Springer.

[12] Freek Wiedijk, editor (2006): *The Seventeen Provers of the World*. *LNAI* 3600, Springer.