CALCULEMUS-2003

11th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning

September 10-12 2003
Roma, Italy

Thérèse Hardin and Renaud Rioboo (Eds)

Foreword

The use of Computer Algebra systems is now wide-spread not only in education or scientific contexts but also in industry, where mathematical software systems help engineers to design systems. In the same way, the growing needs for a more formal approach in software industry require powerful deduction systems, helping engineers to prove that the developments agree with their requirements. The combination of automated mathematical computation and automated mathematical deduction is the major topic of the CALCULEMUS symposium. This includes development of more reliable and accurate computer algebra systems, more powerful and flexible deduction systems. But essentially, the CALCULEMUS symposium is intended to researchers and developers interested in cooperation and unification between the two families of mathematical based software and of their communities. For these reasons, CALCULEMUS symposium co-locate in alternate years with either a Computer Algebra conference or a deduction conferences. This is the case in 2003: CALCULEMUS is co-located with TABLEAUX2003 and TPHOL2003. We thank Marta Cielda, the local organiser of this joined conferences.

We would like to thank the members of the program committee and all the referees for their important work in selecting the submitted papers. We had 29 submissions out of which we selected 6 long papers and 9 short papers. The best papers will be published in a special issue of the London Mathematical Society's Journal of Computation and Mathematics. Submissions will be required

Thérèse Hardin and Renaud Rioboo
Co-chairs

Organization

Program Commitee

Chairs	Thérèse Hardin Renaud Rioboo
Members	Andrea Asperti Henk Barendregt Chris Benzmuller Olga Caprotti James Davenport William Farmer Hoon Hong Fairouz Kamareddine Michael Kohlhase Steve Linton Loic Pottier Roberto Sebastiani Volker Sorge Thomas Sturm Stephen Watt Wolfgang Windsteiger

Additional referees

Philippe Aubry
Gilles Audemard
Quoc Bao Vo
Marco Bozzano
Jacques Carette
Véronique Donzeau-Gouge
Catherine Dubois
Herman Geuvers
Dimitar P Guelev
Manfred Kerber
Temur Kutsia
Roy Mc Casland
Valérie Ménissier-Morain
Milad Niqui
Martin Pollet
Bas Spitters
Jeremie Wajs
Freek Wiedijk
Claus-Peter Wirth

Sponsoring Institutions

http://www.colognet.org/

The European Network of Excellency in Computational Logic

Local Organization

Marta Cialdea Mayer

Table of Contents

The CALCULEMUS Research Training Network — A short Overview*

Christoph Benzmüller

FR Informatik, Universität des Saarlandes, 66041 Saarbrücken, Germany

1 Introduction

This paper sketches the structure and scientific contributions of the CALCULEMUS Research Training Network (CALCULEMUS RTN) since its start in September 2000. It has been reproduced from the networks midterm report [22] and credit is due to all researchers of the CALCULEMUS RTN. More than 28 young visiting researchers (with a sum of approx. 150 financed person-months) have been supported by the network so far and approx. 47 senior researchers are involved in the training measures at the different partner sites. Figure 1 provides the list of the CALCULEMUS RTN partner sites. The network's homepage is `http://www.eurice.de/calculemus/`.

2 Motivation

The long-term motivation of the CALCULEMUS research initiative (see `www.calculemus.net`) is to foster the development of a new generation of assistant systems for mathematics and formal methods. Some key characteristics of the systems CALCULEMUS is aiming at are compiled in the following (incomplete) list:

- Combined support for symbolic reasoning and symbolic computation.
- Interoperability with emerging decentralised and shared mathematical knowledge bases.
- Support mechanisms for the exploration, validation, and maintenance (in particular management of change) of domain specific knowledge.
- Support for flexible integration of heterogeneous specialist reasoners as subsystems (including classical automated theorem provers, model generators, decision procedures, etc.).
- Provision of rich and expressive representation languages and communication means to the users side (probably including rather informal or even natural language based representations) in combination with human-oriented, multimodal user interfaces.

* This work is supported by the EU Research Training Network CALCULEMUS (HPRN-CT-2000-00102) funded in the EU 5th framework.

USAAR	Saarland University, Saarbrücken, Germany (Jörg Siekmann, Christoph Benzmüller)
UED	The University of Edinburgh, Scotland (Alan Bundy)
UKA	Karlsruhe University, Germany (Jacques Calmet)
RISC	Research Institute for Symbolic Computation, Linz, Austria (Bruno Buchberger)
TUE	Eindhoven University of Technology, Netherlands (Arjeh Cohen) University of Nijmegen, Netherlands (Henk Barendregt)
ITC-IRST	Instituto per la Ricerca Scientifica e Tecnologica, Trento, Italy (Fausto Giunchiglia)
UWB	University of Bialystok, Poland (Andrzej Trybulec)
UGE	Università degli Studi di Genova (Alessandro Armando)
UBIR	The University of Birmingham, England (Manfred Kerber)

Fig. 1. The Calculemus RTN

- Support for transformations between the expressive and user-oriented representations employed in the assistant system and the usually highly specialised machine-oriented representations employed by the integrated specialist reasoners.
- Development and utilisation of open system architectures fostering interoperability and tool exchange between different assistant systems (for example, in the emerging mathematical semantic web).
- Direct support for the preparation and validation of mathematical texts and publications.
- Applications in mathematics, mathematics education, and formal methods.

These research goals are ambitious and call for the combination of resources and the mutual exchange of scientific expertise between the involved scientific communities. To tackle them, Calculemus is basically pursuing a bottom-up approach starting from single research aspects as mentioned above and from the existing and emerging tools of the involved research groups.

The current scientific focus is on the integration of symbolic computation and symbolic reasoning which has been identified as a major issue. The sociological goal of the Calculemus RTN is to combine the scientific expertise of the involved researchers in order to optimally train and develop a new generation of young researchers in consideration of the implied scientific challenges.

3 Calculemus RTN: Research Objectives and Results

A predominant research objective of the Calculemus RTN is to foster the integration of deduction systems (DS) and computer algebra systems (CAS), both at a conceptual and at a practical level. The point of origin for this kind of research is a landscape of heterogeneous approaches and systems on both sides of the spectrum, where the diversity on the DSs side is greater than on the side of CASs.

Since its start in September 2000 the CALCULEMUS RTN has contributed to the convergence of DSs and CASs through its research on unifying frameworks for encoding and combining computation and deduction, the identification of the architectural requirements for a new generation of reasoning systems with combined reasoning and computational power, and the prototypical implementation and application of the improved systems. However, a single predominant theoretical framework is currently not possible. Such an approach would particularly involve predominant solutions to the still rather diverging systems at both sides of the spectrum between DSs and CASs. Therefore a strong line of research in the CALCULEMUS RTN focuses on the modelling and integration of CASs and DSs at the systems layer. In this research direction, significant progress has been made and several systems of project partners and other research institutes have been connected in order to form networks of cooperating mathematical service systems. The benefits and impacts of such integrations have been investigated in prototypical case studies.

The researchers of the CALCULEMUS RTN and the CALCULEMUS interest group also fostered the Mathematical Knowledge Management (MKM, EU MKM-NET) research initiative; see [40, 8]. This relatively young line of research adopts a broader perspective on the future of mathematics (e.g. research and publication practice, education, and knowledge maintenance) in the 21st century. A significant amount of CALCULEMUS research is MKM relevant and is currently being taken up in this community in order to adopt and integrate it into the MKM perspective.

The extensive research activities of the CALCULEMUS Network and the CALCULEMUS Interest Group are furthermore shown inter alia by three special issues of the Journal of Symbolic Computation [101, 4, 78] and the following international events: CALCULEMUS Symposium 2000 in St. Andrews, Scotland [69, 101], CALCULEMUS Symposium 2001 in Siena, Italy [78], CALCULEMUS Symposium 2002 in Marseilles, France [45, 49], CALCULEMUS Autumn School 2002 in Pisa, Italy [23–25, 128]. The CALCULEMUS Symposium 2003[1] will be held in September in Rome, Italy, and it will join IJCAR conference in 2004.

In the following paragraphs we sketch the highlights of the CALCULEMUS RTN since its start in September 2000; for more detailed reports to all tasks we refer to [22].

Task 1.1: Mathematical Frameworks TUE and Nijmegen University investigated type theory for the purpose of formalising mathematics: Barendregt and Geuvers [21] give an overview of type theory, how it is used to represent logic and mathematics and what issues and choices come up. Type theory (encoded in OPENMATH) as a way for communicating mathematics is proposed in [20] and in [48] it is shown how a proof presentation can be generated from a formalised proof in type theory. This paper argues that 'formal contexts' in Coq can be used as a basis for interactive mathematical documents. This topic is also

[1] `http://www-calfor.lip6.fr/~rr/Calculemus03/`

treated in [99]. An in-depth discussion of the various ways to treat computations in theorem provers is given in [19] and further related work is presented in [36].

The CALCULEMUS RTN has also studied other approaches to theorem proving and their capacities to integrate computations (see also [122]). This includes proof planning, as developed and employed by the nodes USAAR and UED. In the ΩMEGA system [104], at USAAR, symbolic calculations can be integrated into proof planning in two ways: (i) to guide the proof planner and to prune the search space by computing hints with control rules and (ii) to shorten and simplify the proofs by calling a CAS within the application of a method to solve equations. As a side-effect both cases can restrict possible instantiations of meta-variables. These approaches are discussed in [52, 107, 84, 105].

An investigation into the use of deduction for the implementation of correct computations within computer algebra system was considered at UGE and is presented in [1].

The THEOREMA system, developed at RISC, aims at providing one mathematical framework encompassing all aspects of algorithmic mathematics, notably the aspects of *proving*, *computing*, and *solving*; see [39, 37, 38].

In [70, 71] it is critically argued by UBIR that aspects of mathematical concepts, including procedural knowledge, are hard to reconstruct from the formalisation in deduction systems. This work points to limitations of the flexibility of mathematical representations which apply to all our current approaches.

Task 1.2: Definition of Mathematical Service The primary goal of this Task is the enhancement of existing computer algebra systems and deductive systems by turning them into *open* systems capable of using and/or providing mathematical services. After a preliminary analysis of the state-of-the-art of reasoning systems, it was decided to tackle the problem, in parallel, by a top-down and a bottom-up approach.

In the top-down approach, new infrastructures (both at the conceptual, specification, and architectural level) for the seamless integration of mathematical services have been investigated. This was intended not only for current systems, but also and in particular for future implementations. To this extent particular emphasis was on the definition of frameworks (languages, protocols, semantic specifications, architectural schemata) suitable for making mathematical services accessible over the web. The relevant top-down approaches are: OMRS (Open Mechanised Reasoning Systems) developed by UGE and ITC-IRST [2], LBA (Logic Broker Architecture) developed by UGE [6, 7], MathWeb-SB (MathWeb Software Bus) developed by USAAR [129], MathBroker developed by RISC [81]. These networks can themselves be coupled again as, for instance, exemplarily investigated in [127].

In the bottom-up approach, we have investigated how complex mathematical services can be built out of simpler ones. A particular emphasis has been devoted to decision procedures, and in particular to the integration of procedures specific for solving mathematical problems with deductive procedures. Examples for bottom up approaches are CCR (Constraint Contextual Rewriting) developed by UGE and MathSat [61, 11, 10, 9, 12], developed by ITC-IRST.

In Task 1.2 the CALCULEMUS network also closely cooperates with the EU project MONET. In MONET special ontologies comprising mathematical problems, queries and services have been defined and investigated.

Task 2.1: Integration of CASs and DSs via Protocols Cooperation among several software systems can be achieved with indirect, unidirectional and bidirectional communication. The goal of this task is to investigate how protocols can be defined to provide a semantics as well as soundness results for systems exchanging mathematical information. This definition hints at several other tasks in the CALCULEMUS RTN dealing with very similar problems. This is for example true when defining a context for a computation and is partly covered in Task 1. Unidirectional and bidirectional communication protocols are designed when coupling directly different modules. Although there are no direct links between the services with indirect communication, interaction is possible when systems can communicate with a common user interface, central unit, mediator or evaluator. This approach, which is partly based on a joint work with ITC-IRST on OMSCS (Open Mechanised Symbolic Computation Systems), has been investigated within the KOMET system at UKA see [44, 76, 55, 46].

A semantics can be provided by at least three approaches: (a) define a mathematical software bus, (b) define a context from which a semantic can be derived, (c) formulate the problem as a knowledge representation paradigm.

These approaches are shared by several of the partners. Indeed, they lead to introduce multi-agent systems, contexts, and ontologies to just quote a few features (see for instance the LBA and the MathWeb-SB).

Task 2.2: Enhancing the Reasoning Power of Computer Algebra Systems Enhancement of CAS with reasoning power can be attempted at different levels: (a) enhancement of CAS on the System Level, (b) enhancement of CAS on the Theory Level, and (c) enhancement of CAS on the User Level.

Direction (a) can be achieved by adding additional reasoning capabilities, i.e., logical inference systems, to algorithms built into the CAS. The Constraint Contextual Rewriting (CCR) framework developed by UGE can be used in order to integrate the evaluation mechanism of the CAS MAPLE with an appropriate decision procedure for checking side-conditions, see [1] and [5].

Direction (b) can be achieved by adding proven knowledge about CAS functions to the CAS knowledge base. The HR system, developed at UED, has been used to conjecture properties of functions available in the MAPLE algorithm library from empirical patterns detected in computational data produced by the CAS [53].

Direction (c) can be achieved by giving the CAS user the possibility to prove mathematical statements using proof techniques from logic within the CAS in addition to the computing facilities that each CAS offers. In the framework of the

Calculemus RTN, the work of RISC represents this aspect of CAS enhancement: The Theorema system, see [41], is an add-on package for the widespread and popular CAS *Mathematica* where the user formulates mathematical theorems and proves them entirely within the *Mathematica* environment.

Task 2.3: Enhancing the Computation Power of Deductions Systems UED investigated the combination of the proof-planner $\lambda Clam$ [102] with other systems for computationally costly tasks. This includes (a) an implementation of the GS flexible decision procedure system framework in (Teyjus) LambdaProlog and within the $\lambda Clam$ proof planning system [42] and (b) the integration of the $\lambda Clam$ proof-planner into the MathWeb-SB system [54].

UED also investigated the combination of systems to discover attacks to security protocols [108, 109]. This work makes use of computational power in that it generates a large number of clauses in its processing.

Further relevant work has been done in the $\lambda Clam$ proof-planner to construct very large and modular proof-plans for complicated real analysis theorems [65, 79, 80].

The Ωmega proof planner at USAAR has been coupled with different CASs via MathWeb-SB, see [107, 84, 105]. The Ωants approach to integrate CASs into mathematical assistant systems is sketched in [29, 28, 34, 35]. This work proposes an agent-based modelling of inference rules and external systems at a very basic level within theorem provers.

Finally, work done at UBIR and UGE which render techniques from automated reasoning highly efficient by using enhanced computational power are presented in [66–68] and [9, 12, 3]. Further relevant work is given in [100].

Task 3.1: Automated Support to Writing Mathematical Publications Typically, a mathematical publication contains the following ingredients: natural language text, mathematical formulae, formal text (i.e. definitions and theorems), proofs, examples (typically with computations), and graphics (tables, drawings, sketches, etc.). In the optimal case, a software system for supporting mathematical publications would support all these facets of mathematical publications. Several systems and languages have been used for case studies in this area:

(a) The MIZAR approach (at UWB) is based on two kinds of software which automate the process of writing formal mathematical papers: (i) software used to prepare an article as a formal text whose correctness is computer verified and (ii) the software for automatic (or semi-automatic) translation into natural language (particularly English); this includes also the software for translation into XML-based formats. The cooperation with other Calculemus sites includes development of the MIZAR Mathematical Library (MML) and also the above mentioned translation into XML formats. Relevant publications are [88, 60, 16–18, 94]. Recently published MIZAR articles in the Journal of Formalized Mathematics are [113, 74, 95, 63, 117, 73, 103, 15, 14, 64, 89, 97, 90, 111, 112, 98, 93, 116, 59, 114, 91, 92, 62, 115].

(b) THEOREMA is a prototypical software system designed to give computer-support to the working mathematician during all phases of mathematical activity. Several features qualify THEOREMA as a powerful system for creating mathematical publications entirely inside the system. "Classical" mathematical documents can be written that are intended mainly for printout, as for instance the thesis [125] or the conference papers [123], [124], and [126]. In the case studies, however, emphasis has been put on using the THEOREMA system for developing interactive lecture notes for university mathematics courses. Mostly since the THEOREMA language is very similar to the language used in "ordinary mathematics" the system is highly suitable for this approach, both in illustrating computation-based courses as well as in supporting proof-oriented courses.

(c) The OMDOC [72] content markup scheme which has been developed at USAAR, supports authors with writing formal mathematical documents including articles, textbooks, interactive books and courses. OMDOC allows to capture the semantics and structure of these documents. Various tools are available to transform OMDOC documents into other formats for presentation purposes (using, e.g., MathML) or to support inter-system communication (e.g., by transformation into the logic of a theorem prover).

(d) TUE has developed the MATHDOX tool supporting interactive mathematical documents. MATHDOX is based on DOCBOOK but also has similarities to OMDOC.

Task 3.2: Support to the Development of an Industrial-Strength Application of Formal Methods to Program Verification In addition to formal methods, which is undoubtedly the most important application area for our research, we have identified the education sector as another interesting application for DSs and CASs. Actually the systems THEOREMA (RISC) and ACTIVEMATH [87] (USAAR), which make use of tools and approaches developed in the CALCULEMUS RTN, are already employed in education practice. Another example is the MATHDOX tool developed at TUE since the next version of the interactive textbook *Algebra Interactive!* [51] will appear in this format.

Formal method applications currently pursued in the CALCULEMUS RTN include (a) an approach to support the verification of hybrid systems with the help of mathematical services in MathWeb-SB [27, 26], (b) the investigation whether specialised reasoning tools within the MathWeb-SB can fruitfully support the formal verification of information flow properties and error detection in security protocols [12], and (c) the application of proof planning in first-order linear temporal logic (FOLTL) to feature interactions as they arise in large telephone networks [50].

Task 3.3: Support to the Solution of Undergraduate Exam in Calculus and Economics In this Task we focus on simple, mathematics education oriented problems with a strong emphasis on the particular way the problems are solved, how interaction with the user is supported and how the solution is presented. We analyse whether our systems can be employed in a user friendly and adequate way and whether the interaction and maths presentation capabilities of the systems are appropriate.

A task relevant case pursued at Nijmegen University compares how the problem of proving the irrationality of $\sqrt{2}$, which involves computations, can be proved in fifteen different theorem proving environments (including systems of the CALCULEMUS RTN) [122, 121, 106, 33, 105].

Among the case studies that are currently being started at USAAR are exercises from the German *Bundeswettbewerb Mathematik* and Calculus exercises being encoded and investigated in the ACTIVEMATH project. Empirical studies at USAAR investigates the phenomena of natural language dialog with mathematical assistant systems on proof exercises in naive set theory.

Task 3.4: Modelling of Existing Systems as Mathematical Services The work in this Task so far has concentrated both on developing the required infrastructure (languages, protocols, semantic specifications, architectural schemata) for making existing systems inter-operate, and on studying extensions and enhancements of the reasoning capabilities of some existing tools. The relevant contributions are: (i) MathSat framework developed at ITC-IRST [11, 10], (ii) the RDL (Rewrite and Decision procedure Laboratory), (iii) the LBA [6, 7, 127] developed by UGE, (iv) the modelling of existing systems, for instance, $\lambda Clam$ developed at UED [102], as mathematical services in MathWeb-SB developed at USAAR [54].

Further work at USAAR concentrates on the mediation of mathematical knowledge between the mathematical knowledge base MBASE, which has been integrated to the MathWeb-SB, and mathematical assistant systems such as ΩMEGA [56, 33, 32].

Task 3.5: Challenge Mathematical Problems During the work on the above tasks some challenging mathematical problems had to be tackled already, in order to have non-trivial working examples. Some of the examples were done either by single partner nodes or in collaboration between some of the nodes. The examples include: (i) Fundamental Theorem of Algebra [58, 57], (ii) Involutive Bases [47, 43], (iii) Exploration in Finite Algebra, (iv) The Residue Class Domain [82, 85, 83, 84], (v) Proving with Invariants [86], (vi) The Jordan curve theorem for special polygons, (vii) Continuous lattices [75], (viii) Order sorted algebras [119, 110, 118], (ix) Proofs in Homological Algebra, (x) Proofs in Graph Theory, (xi) Exploration in Zariski Spaces. Further related work is given in [30, 31].

4 Discussion

Prima facie it may appear disappointing that a predominant, single and uniform solution for the integration of deduction and computation is not possible and that the network places an emphasis on integration at the systems level (which requires support for heterogeneous problem representations). However, it is the *authors opinion* that mathematical assistant systems, in particular those for theory exploration, generally have to find a good compromise between a well chosen degree of heterogeneity and flexibility of mathematical representations and the enforcement of representational uniformity. Finding *good* representations has been identified as a key issue in artificial intelligence and the author is convinced that it is important for mathematical theory exploration and mathematics education as well. Unfortunately many of todays deduction system are still strongly afflicted with the spirit of Hilbert's program: the possibility to encode mathematics in a uniform, restrictive manner (e.g. based on set theory) does not imply the usefulness of representational uniformity for theory exploration.

Heterogeneity at the representations and the related systems layer, however, requires support for the semantically validated exchange of information and for transformations of representations (probably including algorithms and proof objects based on them) in various goal directions. For instance, semantical descriptions of system capabilities and uniform information exchange facilities can be used for making heterogeneous systems interoperable in "abstract" level proof development. Transformation mechanisms (if possible and available) may then be later used to generate proof objects in a uniform goal representation format. Alternatively the employed systems may be trusted in the context of their particular usage.

In short, the author claims that a well chosen degree of representational heterogeneity and flexibility should be considered a design requirement for mathematical assistant systems instead of a drawback.

References

1. A. Armando and C. Ballarin. Maple's evaluation process as constraint contextual rewriting. In B. Mourrain, editor, *ISSAC 2001: July 22–25, 2001, University of Western Ontario, London, Ontario, Canada: Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation*, pages 32–37, New York, NY 10036, USA, 2001. ACM Press.
2. A. Armando, A. Coglio, F. Giunchiglia, and S. Ranise. The Control Layer in Open Mechanized Reasoning Systems: Annotations and Tactics. *Journal of Symbolic Computation*, 32(4), 2001.
3. A. Armando, L. Compagna, and S. Ranise. System Description: RDL—Rewrite and Decision procedure Laboratory. In *Automated Reasoning. First International Joint Conference (IJCAR'01), Siena, Italy, June 18–23, 2001, Proceedings*, volume 2083 of *LNAI*, pages 663–669, Berlin, 2001. Springer.
4. A. Armando and T. Jebelean, editors. *Calculemus: Integrating Computation and Deduction*, volume 32 (4) of *Special Issue of Journal of Symbolic Computation on Calculemus'99*, October 2001.
5. A. Armando and S. Ranise. Constraint Contextual Rewriting. *Journal of Symbolic Computation. Special issue on First Order Theorem Proving, P. Baumgartner and H. Zhang editors*, 2002.
6. A. Armando and D. Zini. Towards Interoperable Mechanized Reasoning Systems: the Logic Broker Architecture. In *AI*IA-TABOO Joint Workshop: 'Dagli Oggetti*

agli Agenti: Tendenze Evolutive dei Sistemi Software', pages 70–75, Parma, Italy, 2000. Reprinted in AI*IA Notizie Anno XIII (2000) vol. 3.
7. A. Armando and D. Zini. Interfacing Computer Algebra and Deduction Systems via the Logic Broker Architecture. In Kerber and Kohlhase [69], pages 49–64.
8. A. Asperti, B. Buchberger, and J. H. Davenport, editors. *Mathematical Knowledge Management, Second International Conference, MKM 2003*, Bertinoro, Italy, February 16-18 2003. Springer.
9. G. Audemard, P. Bertoli, A. Cimatti, A. Korniłowicz, and R. Sebastiani. A SAT Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions. In Voronkov [120], pages 195–210.
10. G. Audemard, P. Bertoli, A. Cimatti, A. Korniłowicz, and R. Sebastiani. Efficiently Integrating Boolean Reasoning and Mathematical Solving, 2002. Submitted to Journal of Symbolic Computation.
11. G. Audemard, P. Bertoli, A. Cimatti, A. Korniłowicz, and R. Sebastiani. Integrating Boolean and Mathematical Solving: Foundations, Basic Algorithms and Requirements. In Calmet et al. [45].
12. G. Audemard, A. Cimatti, A. Korniłowicz, and R. Sebastiani. Bounded Model Checking for Timed Systems. In D. A. Peled and M. Y. Vardi, editors, *FORTE 2002: Conference on Formal Techniques for Networked and Distributed Systems*, volume 2529 of *LNCS*, pages 243–259, Houston, Texas, 2002. Springer.
13. M. Baaz and A. Voronkov, editors. *Logic for Programming, Artificial Intelligence, and Reasoning, 9th International Conference, LPAR 2002*, volume 2514 of *LNAI*, Tblisi, Georgia, 2002. Springer.
14. J. Backer and P. Rudnicki. Hilbert basis theorem. *Formalized Mathematics*, 9(**3**):583–589, 2001.
15. J. Backer, P. Rudnicki, and C. Schwarzweller. Ring ideals. *Formalized Mathematics*, 9(**3**):565–582, 2001.
16. G. Bancerek. Development of the theory of continuous lattices in MIZAR. In Kerber and Kohlhase [69].
17. G. Bancerek, N. Endou, and Y. Shidama. Lim-inf convergence and its compactness. *Mechanized Mathematics and Its Applications*, 2(**1**):29–35, 2002.
18. G. Bancerek and P. Rudnicki. A Compendium of Continuous Lattices in MIZAR: Formalizing recent mathematics. *Journal of Automated Reasoning*, 29(**3**):189–224, 2002.
19. H. Barendregt and E. Barendsen. Autarkic computations in formal proofs. *Journal of Automated Reasoning*, 28(3):321–336, 2002.
20. H. Barendregt and A. Cohen. Electronic communication of mathematics and the interaction of computer algebra systems and proof assistants. *Journal of Symbolic Computation*, 32:3–22, 2001.
21. H. Barendregt and H. Geuvers. *Proof Assistants using Dependent Type Systems*, volume 2 of *Handbook of Automated Reasoning*, chapter 18, pages 1149–1238. Elsevier, 2001.
22. C. Benzmüller, editor. *Systems for Integrated Computation and Deduction – Interim Report of the Calculemus IHP Network*, number SR-03-05 in SEKI Report. Saarland University, 2003. `http://www.ags.uni-sb.de/~chris/papers/E5.pdf`.
23. C. Benzmüller and R. Endsuleit, editors. *CALCULEMUS Autumn School 2002: Course Notes (Part I)*, number SR-02-07 in SEKI Report, 2002. `http://www.ags.uni-sb.de/~chris/papers/E2.pdf`.
24. C. Benzmüller and R. Endsuleit, editors. *CALCULEMUS Autumn School 2002: Course Notes (Part II)*, number SR-02-08 in SEKI Report, 2002. `http://www.ags.uni-sb.de/~chris/papers/E3.pdf`.

25. C. Benzmüller and R. Endsuleit, editors. *CALCULEMUS Autumn School 2002: Course Notes (Part III)*, number SR-02-09 in SEKI Report, 2002. `http://www.ags.uni-sb.de/~chris/papers/E4.pdf`.
26. C. Benzmüller, C. Giromini, and A. Nonnengart. Symbolic Verification of Hybrid Systems supported by Mathematical Services. In Caprotti and Sorge [49]. Seki-Report Series Nr. SR-02-04, Universität des Saarlandes.
27. C. Benzmüller, C. Giromini, A. Nonnengart, and J. Zimmer. Reasoning services in the mathweb-sb for symbolic verification of hybrid systems. In *Proceedings of the Verification Workshop - VERIFY'02 in connection with FLOC 2002*, pages 29–39, Kopenhagen, Denmark, 2002.
28. C. Benzmüller, M. Jamnik, M. Kerber, and V. Sorge. An Agent-oriented Approach to Reasoning. In Linton and Sebastiani [77].
29. C. Benzmüller, M. Jamnik, M. Kerber, and V. Sorge. Experiments with an Agent-oriented Reasoning System. In *KI 2001: Advances in Artificial Intelligence*, Vienna (Austria), 2001.
30. C. Benzmüller and M. Kerber. A Challenge for Automated Deduction. In *Proceedings of IJCAR-Workshop: Future Directions in Automated Reasoning*, Siena (Italy), 2001.
31. C. Benzmüller and M. Kerber. A Lost Proof. In *TPHOLs: Work in Progress Papers*, Edinburgh (Scotland), 2001.
32. C. Benzmüller, A. Meier, and V. Sorge. Distributed assertion retrieval. In *First International Workshop on Mathematical Knowledge Management RISC-Linz*, pages 1–7, Schloss Hagenberg, 2001.
33. C. Benzmüller, A. Meier, and V. Sorge. Bridging Theorem Proving and Mathematical Knowledge Retrieval. In D. Hutter and W. Stephan, editors, *Festschrift in Honour of Prof. Jörg Siekmann*, LNAI. Springer, 2003. To appear.
34. C. Benzmüller and V. Sorge. Oants – an open approach at combining interactive and automated theorem proving. In Kerber and Kohlhase [69], pages 81–97.
35. C. Benzmüller and V. Sorge. Agent-based Theorem Proving. In *9th Workshop on Automated Reasoning*, London (GB), March 2002.
36. A. Bove and V. Capretta. Nested general recursion and partiality in type theory. In R. J. Boulton and P. B. Jackson, editors, *Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2001.
37. B. Buchberger. Theorema: A short introduction. *Mathematica Journal*, 8(2):247–252, 2001.
38. B. Buchberger. Theorema: Extending mathematica by automated proving. In D. Ungar, editor, *Proceedings of PrimMath 2001 (The Programming System Mathematica in Science, Technology, and Education)*, pages 10–11, University of Zagreb, Electrotechnical and Computer Science Faculty, September 27-28 2001.
39. B. Buchberger, C. Dupré, T. Jebelean, K. Kriftner, K. Nakagawa, D. Vasaru, and W. Windsteiger. The *Theorema* Project: A Progress Report. In Kerber and Kohlhase [69].
40. B. Buchberger, G. Gonnet, and M. Hazewinkel, editors. *Mathematical Knowledge Management (MKM 2001) – Special issue of Annals in Mathematics and Artificial Intelligence,*. Kluwer, 2003. To appear.
41. B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuta, and D. Vasaru. A survey of the *theorema* project. In W. Kuechlin, editor, *Proceedings of ISSAC'97 (International Symposium on Symbolic and Algebraic Computation*, pages 384–391, Maui, Hawaii, July 1997. ACM Press.

42. A. Bundy and P. Janičić. A General Setting for Flexibly Combining and Augmenting Decision Procedures. *Journal of Automated Reasoning*, 3(28), 2002.
43. J. Calmet. Intas: Final report. Internal Report: http://iaks-www.ira.uka.de/iaks-calmet/intas.html, 2002.
44. J. Calmet, C. Ballarin, and P. Kullmann. Integration of deduction and computation. *Applications of Computer Algebra*, pages 15–32, 2001.
45. J. Calmet, B. Benhamou, O. Caprotti, L. Henocque, and V. Sorge, editors. *CALCULEMUS-2002: Symposium on the Integration of Symbolic Computation and Mechanized Reasoning*, volume 2385 of *LNAI*. Springer, 2002.
46. J. Calmet, F. Freitas, and G. Bittencourt. Master-web: An ontology-based internet data mining multi-agent system. In *Proceedings of SSGRR 2001, Computer & e-Business conference*, 2001.
47. J. Calmet, W. Hausdorf, and W. Seiler. A constructive introduction to involution. In R. Akerkar, editor, *Proc. Int. Symp. Applications of Computer Algebra - ISACA 2000*, pages 33–50. Allied Publishers Limited, 2001.
48. O. Caprotti, H. Geuvers, and M. Oostdijk. Certified and portable mathematical documents from formal contexts. In B. Buchberger and O. Caprotti, editors, *MKM 2001 (1st International Workshop on Mathematical Knowledge Management)*, Research Institute for Symbolic Computation, Johannes Kepler University, Hagenberg, September 24-26 2001.
49. O. Caprotti and V. Sorge, editors. *Calculemus 2002, 10th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning: Work in Progress Papers*, Marseilles, France, June 2002. Seki-Report Series Nr. SR-02-04, Universität des Saarlandes.
50. C. Castellini and A. Smaill. Proof planning for feature interactions: a preliminary report. In Baaz and Voronkov [13].
51. A. Cohen, H. Cuypers, and H. Sterk. *Algebra Interactive!* Springer, 1999.
52. A. Cohen, S. Murray, M. Pollet, and V. Sorge. Certifying solutions to permutation group problems. Submitted to a major international conference, 2003.
53. S. Colton. Making conjectures about maple functions. In Calmet et al. [45].
54. L. Dennis and J. Zimmer. Inductive theorem proving and computer algebra in the mathweb software bus. In Calmet et al. [45].
55. R. Endsuleit and T. Mie. Protecting co-operating mobile agents against malicious hosts. Internal Report 2002-8, University of Karlsruhe, 2002.
56. A. Franke, M. Moschner, and M. Pollet. Cooperation between the Mathematical Knowledge Base MBase and the Theorem Prover Omega. In Caprotti and Sorge [49]. Seki-Report Series Nr. SR-02-04, Universität des Saarlandes.
57. H. Geuvers, R. Pollack, F. Wiedijk, and J. Zwanenburg. A constructive algebraic hierarchy in coq. *Journal of Symbolic Computation*, 34(4):271–286, 2002.
58. H. Geuvers, F. Wiedijk, and J. Zwanenburg. A constructive proof of the fundamental theorem of algebra without using the rationals. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs, Proceedings of the International Workshop, TYPES 2000, Durham*, number 2277 in LNCS, pages 96–111. Springer, 2001.
59. M. Giero. On the general position of special polygons. *Formalized Mathematics*, 10(**2**):89–95, 2002.
60. G. Gierz, K. Hofmann, K. Keimel, J. Lawson, M. Mislove, and D. Scott. *A Compendium of Continuous Lattices*. Springer-Verlag, Berlin, Heidelberg, New York, 1980.
61. F. Giunchiglia, R. Sebastiani, and P. Traverso. Integrating SAT solvers with domain-specific reasoners. In Kerber and Kohlhase [69].

62. A. Grabowski. On the decompositions of intervals and simple closed curves. *Formalized Mathematics*, 10(**3**):145–151, 2002.
63. A. Grabowski, A. Korniłowicz, and A. Trybulec. Some properties of cells and gauges. *Formalized Mathematics*, 9(**3**):545–548, 2001.
64. E. Grądzka. The algebra of polynomials. *Formalized Mathematics*, 9(**3**):637–643, 2001.
65. A. Heneveld, E. Maclean, A. Bundy, A. Smaill, and J. Fleuriot. Towards a formalisation of college calculus. In Kerber and Kohlhase [69].
66. M. Jamnik, M. Kerber, and M. Pollet. Automatic learning in proof planning. Technical Report CSRP-02-3, University of Birmingham, School of Computer Science, March 2002.
67. M. Jamnik, M. Kerber, and M. Pollet. Automatic learning in proof planning. In F. van Harmelen, editor, *ECAI-2002: European Conference on Artificial Intelligence*, pages 282–286. IOS Press, 2002.
68. M. Jamnik, M. Kerber, and M. Pollet. LearnOmatic: System description. In Voronkov [120], pages 150–155.
69. M. Kerber and M. Kohlhase, editors. *Symbolic Computation and Automated Reasoning – The CALCULEMUS-2000 Symposium*, St. Andrews, UK, August 6–7, 2000 2001. AK Peters, Natick, MA, USA.
70. M. Kerber and M. Pollet. On the design of mathematical concepts. Cognitive Science Research Papers CSRP-02-06, The University of Birmingham, School of Computer Science, May 2002.
71. M. Kerber and M. Pollet. On the design of mathematical concepts. In B. McKay and J. Slaney, editors, *AI-2002: 15th Australian Joint Conference on Artificial Intelligence*. Springer, LNAI, 2002.
72. M. Kohlhase. OMDoc: Towards an internet standard for the administration, distribution and teaching of mathematical knowledge. In *Proceedings of AI and Symbolic Computation, AISC-2000*, LNAI. Springer Verlag, 2000.
73. A. Korniłowicz and R. Milewski. Gauges and cages. Part II. *Formalized Mathematics*, 9(**3**):555–558, 2001.
74. A. Korniłowicz, R. Milewski, A. Naumowicz, and A. Trybulec. Gauges and cages. Part I. *Formalized Mathematics*, 9(**3**):501–509, 2001.
75. J. Kotowicz and Y. Nakamura. Go-board theorem. *Formalized Mathematics*, 3(**1**):125–129, 1992.
76. P. Kullmann. *Wissensrepraesentation und Anfragebearbeitung in einer logikbasierten Mediatorumgebung*. PhD thesis, University of Karlsruhe, 2001.
77. S. Linton and R. Sebastiani, editors. *CALCULEMUS-2001 – 9th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning*, Siena, Italy, June 21–22 2001.
78. S. Linton and R. Sebastiani, editors. *Journal of Symbolic Computation, Special Issue on the Integration of Automated Reasoning and Computer Algebra Systems*, volume 34 (4). Elsevier, 2002.
79. E. Maclean. Automating proof in non-standard analysis (ii). In *Proceedings of ESSLLI 2001*, Helsinki, 2001.
80. E. Maclean, J. Fleuriot, and A. Smaill. Proof-planning non-standard analysis. In *Proceedings of the 7th International Symposium on Aritifical Intelligence and Mathematics*, Fort Lauderdale, 2002.
81. Mathbroker - A Framework for Brokering Distributed Mathematical services. `http://poseidon.risc.uni-linz.ac.at:8080/index.html`.

82. A. Meier, M. Pollet, and V. Sorge. Exploring the Domain of Residue Classes. Seki Report SR-00-04, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, Germany, December 2000.
83. A. Meier, M. Pollet, and V. Sorge. Classifying Isomorphic Residue Classes. In Moreno-Díaz et al. [96], pages 494–508.
84. A. Meier, M. Pollet, and V. Sorge. Comparing Approaches to the Exploration of the Domain of Residue Classes. *Journal of Symbolic Computation, Special Issue on the Integration of Automated Reasoning and Computer Algebra Systems*, 34(4):287–306, 2002.
85. A. Meier and V. Sorge. Exploring Properties of Residue Classes. In Kerber and Kohlhase [69], pages 175–190.
86. A. Meier, V. Sorge, and S. Colton. Employing theory formation to guide proof planning. In Calmet et al. [45], pages 275–289.
87. E. Melis, E. Andres, J. Büdenbender, A. Frischauf, G. Goguadze, P. Libbrecht, M. Pollet, and C. Ullrich. Activemath: A generic and adaptive web-based learning environment. *Journal of Artificial Intelligence and Education*, 12(4):385–407, 2001.
88. R. Milewski. Fundamental theorem of algebra. *Formalized Mathematics*, 9(**3**):461–470, 2001.
89. R. Milewski. Upper and lower sequence of a cage. *Formalized Mathematics*, 9(**4**):787–790, 2001.
90. R. Milewski. Upper and lower sequence on the cage. Part II. *Formalized Mathematics*, 9(**4**):817–823, 2001.
91. R. Milewski. Properties of the internal approximation of Jordan's curve. *Formalized Mathematics*, 10(**2**):111–115, 2002.
92. R. Milewski. Properties of the upper and lower sequence on the cage. *Formalized Mathematics*, 10(**3**):135–143, 2002.
93. R. Milewski. Upper and lower sequence on the cage, upper and lower arcs. *Formalized Mathematics*, 10(**2**):73–80, 2002.
94. R. Milewski and C. Schwarzweller. Algebraic requirements for the construction of polynomial rings. *Mechanized Mathematics and Its Applications*, 2:1–8, 2002.
95. R. Milewski, A. Trybulec, A. Korniłowicz, and A. Naumowicz. Some properties of cells and arcs. *Formalized Mathematics*, 9(**3**):531–535, 2001.
96. R. Moreno-Díaz, B. Buchberger, and J.-L. Freire, editors. *Proceedings of the 8th International Workshop on Computer Aided Systems Theory (EuroCAST 2001)*, volume 2178 of *LNCS*, Las Palmas de Gran Canaria, Spain, February 19–23 2001. Springer Verlag, Berlin, Germany.
97. A. Naumowicz. Some remarks on finite sequences on go-boards. *Formalized Mathematics*, 9(**4**):813–816, 2001.
98. A. Naumowicz and R. Milewski. Some remarks on clockwise oriented sequences on go-boards. *Formalized Mathematics*, 10(**1**):23–27, 2002.
99. M. Oostdijk. *Generation and Presentation of Formal Mathematical Documents.* PhD thesis, Eindhoven University of Technology, Sept. 2001.
100. S. Ranise. Combining generic and domain specific reasoning by using contexts. In Calmet et al. [45].
101. T. Recio and M. Kerber, editors. *Computer Algebra and Mechanized Reasoning: Selected St. Andrews' ISSAC/Calculemus 2000 Contributions*, volume 32(1/2) of *Journal of Symbolic Computation*, 2001.
102. J. D. C. Richardson, A. Smaill, and I. Green. System description: proof planning in higher-order logic with Lambda-Clam. In *CADE'98*, volume 1421 of *LNCS*, pages 129–133, 1998.

103. C. Schwarzweller. The binomial theorem for algebraic structures. *Formalized Mathematics*, 9(**3**):559–564, 2001.
104. J. Siekmann, C. Benzmüller, V. Brezhnev, L. Cheikhrouhou, A. Fiedler, A. Franke, H. Horacek, M. Kohlhase, A. Meier, E. Melis, M. Moschner, I. Normann, M. Pollet, V. Sorge, C. Ullrich, C.-P. Wirth, and J. Zimmer. Proof development with omega. In Voronkov [120], pages 144–149.
105. J. Siekmann, C. Benzmüller, A. Fiedler, A. Meier, and M. Pollet. Irrationality of Square Root of 2 - A Case Study in OMEGA. Submitted to an International Journal, 2002.
106. J. Siekmann, C. Benzmüller, A. Fiedler, A. Meier, and M. Pollet. Proof development with omega: Sqrt(2) is irrational. In Baaz and Voronkov [13], pages 367–387.
107. V. Sorge. Non-Trivial Symbolic Computations in Proof Planning. In H. Kirchner and C. Ringeissen, editors, *Proceedings of Third International Workshop Frontiers of Combining Systems (FROCOS 2000)*, volume 1794 of *LNCS*, pages 121–135, Nancy, France, March 22–24 2000. Springer Verlag, Berlin, Germany.
108. G. Steel, A. Bundy, and E. Denney. Finding counterexamples to inductive conjectures and discovering security protocol attacks. *AISB Journal*, 1(2), 2002.
109. G. Steel, A. Bundy, and E. Denney. Finding counterexamples to inductive conjectures and discovering security protocol attacks. In *Proceedings of the Foundations of Computer Security Workshop*, 2002. Appeared in Proceedings of The Verify'02 Workshop as well. Also available as Informatics Research Report EDI-INF-RR-0141.
110. A. Trybulec. Many sorted algebras. *Formalized Mathematics*, 5(**1**):37–42, 1996.
111. A. Trybulec. More on the external approximation of a continuum. *Formalized Mathematics*, 9(**4**):831–841, 2001.
112. A. Trybulec. More on the finite sequences on the plane. *Formalized Mathematics*, 9(**4**):843–847, 2001.
113. A. Trybulec. Some lemmas for the jordan curve theorem. *Formalized Mathematics*, 9(**3**):481–484, 2001.
114. A. Trybulec. Introducing spans. *Formalized Mathematics*, 10(**2**):97–98, 2002.
115. A. Trybulec. On the minimal distance between sets in Euclidean space. *Formalized Mathematics*, 10(**3**):153–158, 2002.
116. A. Trybulec. Preparing the internal approximations of simple closed curves. *Formalized Mathematics*, 10(**2**):85–87, 2002.
117. A. Trybulec and Y. Nakamura. Again on the order on a special polygon. *Formalized Mathematics*, 9(**3**):549–553, 2001.
118. J. Urban. Free order sorted universal algebra. *Formalized Mathematics*, 10(**3**):211–225, 2002.
119. J. Urban. Order sorted algebras. *Formalized Mathematics*, 10(**3**):179–188, 2002.
120. A. Voronkov, editor. *Proceedings of the 18th International Conference on Automated Deduction (CADE-19)*, volume 2392 of *LNAI*, Copenhagen, Denmark, 2002. Springer.
121. F. Wiedijk. The fifteen provers of the world. Unpublished Draft available at `http://www.cs.kun.nl/~freek/notes/index.html`.
122. F. Wiedijk. Comparing mathematical provers. In Asperti et al. [8].
123. W. Windsteiger. Building up hierarchical mathematical domains using functors in *mathematica*. In A. Armando and T. Jebelean, editors, *Calculemus 99: International Workshop on Combining Proving and Computation*, volume 23(3) of *Electronic Notes in Theoretical Computer Science*, pages 83–102, Trento, Italy, 1999. Elsevier. CALCULEMUS 99 Workshop, Trento, Italy.

124. W. Windsteiger. A Set Theory Prover in Theorema. In Moreno-Díaz et al. [96], pages 525–539. extended version available as RISC report 01-07.
125. W. Windsteiger. *A Set Theory Prover in Theorema: Implementation and Practical Applications*. PhD thesis, RISC Institute, May 2001.
126. W. Windsteiger. On a Solution of the Mutilated Checkerboard Problem using the Theorema Set Theory Prover. In Linton and Sebastiani [77].
127. J. Zimmer, A. Armando, and C. Giromini. Towards Mathematical Agents – Combining MathWeb-SB and LB. In Linton and Sebastiani [77], pages 64–77.
128. J. Zimmer and C. Benzmüller, editors. *CALCULEMUS Autumn School 2002: Student Poster Abstracts*, number SR-02-06 in SEKI Report, 2002.
129. J. Zimmer and M. Kohlhase. System Description: The MathWeb Software Bus for Distributed Mathematical Reasoning. In Voronkov [120], pages 144–149.

Querying Distributed Digital Libraries of Mathematics

Ferruccio Guidi* and Claudio Sacerdoti Coen*

Department of Computer Science
Mura Anteo Zamboni 7, 40127 Bologna, ITALY.
{fguidi,sacerdot}@cs.unibo.it

Abstract. Several of the most effective techniques to query libraries of structured mathematics are based on pattern-matching. Among them, there are matching-based queries, unification-based queries and several kinds of queries up to isomorphisms (as associative-commutative rewriting). All of these techniques do not scale when the libraries become large or when they are distributed. In this paper we present a filtering technique that can be applied to locate a set of candidates that are likely to match the given pattern. Thus our technique can be used as a first phase that is followed by the actual matching over the set of candidates. Tests performed over the library of the Coq system (`http://coq.inria.fr`) (about 40.000 theorems) show that the candidates can be located rather quickly and that the number of false matches is surprisingly low.

1 Introduction

As Jorge Luis Borges pointed out in his beautiful novel "The Library of Babel" [8], a huge library is completely useless unless we have effective methods to retrieve what we are looking for. Unfortunately, the largest part of the mathematical documents available over the Web are Postscripts or PDF documents. Thus, the only reasonable way to look for a given mathematical definition or result is to rely on textual matching or to perform queries over the provided metadata (when available). In particular, the organizations of librarians have developed several useful classification schemes to index mathematical textbooks and papers [4, 1, 2]. This approach can be effective enough when we are looking for a general work on a given mathematical topic; but it can not be used to retrieve a single theorem or definition over the Web, especially when the theorem is not fundamental enough to be given a well-known name.

The forthcoming development of Web services for automated deduction and computation put high pressure for the development of alternative effective methods to retrieve a single result. Luckily, at the same time more and more mathematical knowledge is put on the web in a structured form, thanks to the introduction of standard markup languages [6, 3, 5] for the encoding of mathematical formulae and expressions. Once the statements in a library are marked up, say,

* Supported by MOWGLI: European FET Project IST-2001-33562.

in OpenMath, we can use standard pattern-matching or unification techniques to retrieve a theorem given a part of its statement. For example, the pattern $\forall x : ?_1.\ \forall y : ?_1.\ (?_2\ x\ y) = (?_2\ y\ x)$ will retrieve any theorem stating the commutative property of a binary operation. The question marks are *metavariables*, which are non-linear placeholders for any (well-typed, if enforceable) subexpression closed in the context.

To resolve a query based on pattern matching, the trivial approach consists in iterating the matching operation over the whole library. This solution does not scale well and it can not be applied in those cases where the library itself is physically distributed and no code can be run on the severs[1].

To solve this problem, we propose an approach based on four distinct phases:

- *Data-mining*: using a spider, we extract a small set of automatically computed metadata from each theorem or definition in the distributed library. These metadata are devised to capture some of the invariants of the chosen matching operation.
- *Pattern compilation:* when the user submits a pattern, we extract a set of constraints over these metadata from the pattern and we generate a low-level query[2] starting from these constraints.
- *Filtering:* we execute the generated query over the database built by the spider, obtaining our set of candidates.
- *Matching:* we iterate the matching operation only on the set of candidates.

If the filtering operation is both quick and correct (in the sense that no good candidates are dropped), we are able to achieve both accuracy and performance.

Since the metadata must capture the invariants of matching, we need to identify several kinds of metadata. In sections 2–4 we will progressively consider several examples of matching criteria and we will identify the related metadata. We will also consider the issue of correctness and a tradeoff between correctness and performance. In section 5 we will present a generic set of metadata and constraints that can subsume all the previous examples. Finally, section 6 and section 7 are devoted to comparisons with other works and future developments.

2 Use Case 1: Finding the Applicable Theorems

The first use case that we consider is knowing which theorems can be applied to prove a given goal. A goal is a sequent made of several hypotheses and a conclusion. A theorem can be applied if its conclusion, where every bound variable has been replaced with a metavariable, matches the conclusion of the sequent.

[1] Indeed, if the library is distributed, we have only two possibilities: either we can perform the matching where the theorem is located or we need to download it. In the second case the download times make the operation unfeasible already for very small libraries. In the first case we need to install software were the library is published, which is something we would rather avoid.

[2] In our prototype, we generated the query in the MathQL Level 1 language [12]. Generating a query in other languages is surely possible.

We restrict ourselves to the case of *first order matching*, i.e. only one expression (the conclusion of the theorem) has metavariables in it; and the expression is rigid. For example:

- The theorem $\forall n.\ \forall m.\ \forall a.\ a \geq 0 \Rightarrow n \leq m \Rightarrow an \leq am$ can be applied to prove $2x \leq 2(x+1)$ since the pattern $?a?n \leq ?a?m$ matches $2x \leq 2(x+1)$.
- The theorem $\forall n.\ \forall m.\ \forall a. n \leq m \Rightarrow an \leq am$ can not be applied to prove $x \leq 2x$ since the pattern $?a?n \leq ?a?m$ does not match $x \leq 2x$.
- The induction principle over natural numbers $\forall P.\ (P\ 0) \Rightarrow (\forall n.\ (P\ n) \Rightarrow (P\ (n+1))) \Rightarrow \forall n.\ (Pn)$ does not match $(is\_prime\ 5)$ since the pattern $(?P\ ?n)$ is not rigid.

Now we identify a set of invariants of the matching procedure that we want to capture as metadata:

- The constant in *head position* in the conclusion of the theorem. Since the pattern is rigid, any matching term must have the same head constant of the pattern.
- The list of constants in the conclusion. Since we are doing matching and not unification, any constant in the pattern must also occur in the matching term (since instantiating a metavariable can not remove an occurring constant).

Thus we identify the following set of metadata: for each theorem in the library we remember the name of the constant in head position in the conclusion and the names of all the constants occurring in the rest of the conclusion. Now, to find the theorems that are candidates for the query "give me all the theorems that can be applied to this goal", we simply compute these metadata over the conclusion of the goal and we look for every theorem in the library such that:

- the "constant in head position" metadata is the same as the computed one
- the set of "constant in conclusion" metadata is a subset of the computed one

The following theorem is a trivial consequence of the fact that our metadata reflects the matching invariants:

Theorem 1. *The filtering operation is correct, i.e. no pattern that matches the goal conclusion is filtered out.*

Let's see how it works in practice: given the goal $2x \leq 2(x+1)$

- $?a?n \leq ?a?m$ is a candidate that will match
- $?n \leq ?n^2$ is not a candidate, since the power operator does not occur in the goal
- $?n?a \leq ?m?a$ is a candidate that will not match (a *false match*)

We could also capture a much stronger invariant, that subsumes both our previous invariants: *the rigid skeleton of the term is preserved by metavariable substitutions.* Nevertheless, the stronger invariant has two drawbacks: the first one is that it produces complex metadata (contexts, i.e. terms with holes) that can not be handled efficiently within the relational database model or the RDF model (which is often reduced to the relational model); the second drawback is that the invariant is probably too strong. In particular, it would reject the third candidate of the previous example. Even if that candidate turns out to be a false match, it is pretty obvious (at least to a human user) that the statement found is useful to proceed in the proof. Very often, indeed, the user is also interested in theorems that can "almost" be applied, since they can reveal an error in the proof (when a required assumption is missing) or they can suggest a different way to prove the goal.

Of course, the danger of not adopting the stronger invariant is that of finding too many useless false matches. To evaluate our invariants, we tried several queries over the library of the Coq proof-assistant (about 40.000 theorems). The queries were not artificial: one of the authors used the query engine in order to retrieve some useful lemmas while developing a new medium-sized Coq contribution. Even for queries involving only frequently used notions (e.g. algebraic operations), the accuracy of the filtering phase was always very high: the number of uninteresting false matches was very low and in more than one occasion the search engine found false matches that we were actually able to exploit. Thus, the productivity level of the author was definitely raised by the use of the search-engine.

Let's face now the issue of performance, considering for instance the goal $2x \leq 2(x+1)$ which generates the constraints:

- $\leq$ `in head position in the conclusion`
- `only` 2 `and` + `in other positions in the conclusion`

The generated query (in a pseudo-language inspired by MathQL level 1 [12]) is:

```
select every t in the library such that
 t.head = '<=' and t.in_conclusion subset of {'2','+'}
```

Since the second test can not be performed in a relational DB, we are forced to extract from the DB all the theorems `t` that satisfy the first constraint together with their `in_conclusion` field and then iterate the second test. Thus, the computational cost is linear over the size of the result set of the first test. If the head position of the goal is a frequently used constant, the result set is large. For example, there is an equality in head position in the 11% of the theorems of the library of Coq, i.e. in 3660 theorems. To reduce the computational cost of the query, we propose a user-controlled tradeoff with the accuracy of the query: the user provides an additional set of constraints consisting in a list of constants that *must* appear in the conclusion of the matched theorems. To avoid confusion, we will call these constraints *must constraints* and we rename the previous set of constraints *only constraints*. In order to retrieve a non-empty subset of results, the set of constants occurring in the *must constraints* must be a subset of the constants occurring in the *only constraints*.

To better understand the *must constraints*, let us consider the goal $2x \leq 2(x+1)$. The generated *only constraints* are $\leq$ in head position and 1,2,$*$ and $+$ in other positions in the conclusion. Let's suppose that the user choice of *must constraints* is $\leq$ in head position and $*$ in other positions. Then

– $?a?n \leq ?a?m$ is a candidate that will match (a true match)
– $?n?a \leq ?m?a$ is a candidate that will not match (an interesting false match)
– $?n \leq ?n + 1$ is not a candidate since it satisfies the *only constraints* but not the *must constraints* ('$*$' does not occur). Anyway, it would have been an uninteresting false match.

Note that the *must constraints* are meaningful to the user: in the previous example, the *must constraints* are stating that the user is interested in a property of multiplication. Thus, it is not difficult to provide an user-friendly interface to select these constraints. It is also easy to provide some heuristics to automatically chose the *must constraints*[3].

A non-empty set of *must constraints* reduces the accuracy of the query, since some theorems that could have been applied are not found any longer. For example, the previous query is unable to find the transitivity principle for the $\leq$ relation, since it states nothing about multiplication[4]. The performance gain achieved can be remarkable. To understand why, let us see the query generated for the last example:

```
let S =
 select every t in the library such that
  t.head = '<=' and '*' occurs in t.in_conclusion
in
 select every t in S such that
  t.in_conclusion subset of {'1','2','+','*'}
```

The first part of the query (the first `select`) can be performed by a single call to the database. Then, as before, we need to iterate the second test over the set returned by the first select. The more *must constraints* we have, the smallest the set `S` will be and the smallest the time required to perform the query will be.

In the next two sections we will try to apply the same approach to another couple of examples, choosing the matching operation we are interested in and deriving the *must* and *only constraints* from it.

3 Use Case 2: Finding the Elimination Principles

Let us consider queries for the retrieval of all the elimination principles "à la McBride" [13] over a given datatype. Briefly, these are elimination principles that allow to prove a generic property P over an element n of a datatype T by proving that P holds in several cases, some of them under additional inductive hypotheses. For example, there are several elimination principles over lists of natural numbers. Among them, we find the structural induction principle $\forall P.\ (P\ \mathit{empty}) \Rightarrow (\forall l.\ \forall n.\ (P\ l) \Rightarrow (P\ n :: l)) \Rightarrow \forall l.\ (P\ l)$ and the

[3] As an example, we provide a predefined heuristic that chooses only the constants occurring in the first n levels of the syntactic tree of the goal thesis.

[4] In our experience, very often loosing these "very general" theorems can be considered a feature.

induction principle over ordered lists $\forall P.\ (P\ empty) \Rightarrow (\forall l.\ \forall n.\ (ordered\ l) \Rightarrow n \leq (head\ l) \Rightarrow (P\ n :: l)) \Rightarrow \forall l.\ (ordered\ l) \Rightarrow (P\ l)$.

Note that, since the pattern $(?P\ ?x)$ is not rigid, every elimination principle can always be used to progress in any proof, independently of the goal. Since there exist literally thousands of elimination principles, it is not a good idea to include them as results of the query of the first use case we considered[5]. Nevertheless, we are interested in giving the user the possibility to retrieve all elimination principles over a given datatype. In particular, given a datatype T, we want to retrieve all the theorems whose statement matches[6] the schema:

$$\forall \overline{x} : \overline{S}.\ \forall P : T \rightarrow Prop.\ \forall \overline{y} : \overline{S'}.\ \forall x : T. \forall \overline{z} : \overline{S''}.\ (P\ x)$$

We identify the following set of constraints:

- The sort `Prop` (i.e. the type of every proposition) must occur in the head position of an hypothesis which is a product[7] of length 1.
- The datatype must occur in the head position of an hypothesis which is a product of length 0 and and also in another hypothesis (not in main position).
- The head of the conclusion must be an occurrence of a bound variable.

Every constraint captures a matching invariant, since context metavariables substitution can not change the shape of the conclusion nor the shape of an hypothesis (which are all rigid parts). As in the previous use-case, there exist stricter invariants: the set of constraints identified here has been fine-tuned by hand to be able to retrieve also a few useful false matches. Even in this case, though, the final set of invariants was identified quite soon and without any major effort.

Note that, unlike the previous use case, we derive the matching pattern from the user input and not from the statements of the theorems in the library. In section 2 we had just one term to match against several different patterns; here we have just one pattern to match against the theorems in the library. This duality is reflected in the way the constraints are used to generate the query: in the previous case the identified constraints were *only constraints* (only the constants occurring in the goal can occur in the statement of the theorem); in this case the constraints are *must constraints* (all the constants in the pattern must occur in the statement of the theorem)[8].

---

[5] That is the reason why we constrained ourselves to first order unification in section 2.

[6] We do not specify formally here the kind of match we are interested in. Anyway, note that the "vector parts" of the schema play the role of *context metavariables*, in the sense that they are contexts of formulae that must be instantiated with a context made of repeated quantifications or implications, concluded by a single contextual hole.

[7] The terminology comes from type-theory and is derived by the Curry-Howard isomorphism, where universal quantifications are seen as (dependent) products. A product of length n is a term of the form $\forall x_1.\ \ldots\ \forall x_n.\ t$ where t does not start with an universal quantification. An implication is just a special case of universal quantification: it is a non-dependent quantification where the bound variable does not occur in the sub-expression.

[8] Both sentences are instances of the following invariant that is true for any kind of matching: "all the constants in the pattern must occur in the matched term".

The query generated from the constraints is the following:

```
select every t in the library such that
 Prop occurs in head position at depth 1 in an hypothesis of t and
 T occurs in head position at depth 0 in an hypothesis of t and
 T occurs not in head position in an hypothesis of t and
 a bound variable occurs in head position in the conclusion of t
```

Unlike the case of section 2, the generated query is already very efficient: this time we do not need any trade-off to improve the performance.

4 Use Case 3: Finding the Proofs of a Statement

The last use case we present is locating a proof (or a definition) whose statement (i.e. its type) is known. The query seems quite uninteresting, but the inability to perform it may have very bad implications in terms of waste of efforts. Indeed, every time we need to prove a lemma, we would like to issue a query that checks if somebody else already proved it; the Coq system provides no way to perform this operation and, as a result, it is not at all unusual to find simple arithmetical properties proved five, eight or even more times by different persons working in remotely located teams[9].

The kind of matching we are interested in is extremely simple: no metavariables are present in both the statement of the theorem and the statements in the library. Close enough false matches, though, play an important role: for example, when looking for a proof of $\forall n.\, n = n * 1$, any of the following statements would be satisfactory: $\forall n.\, n = 1 * n$; $\forall n.\, n * 1 = n$; $\forall n. 1 * n = n$

We identify the following set of metadata, which constitute a close approximation of the first two levels of the abstract syntax tree of the pattern:

- For each occurrence of a constant, we record its *position*, as an element of the following set: {`MainConclusion, InConclusion, MainHypothesis, InHypothesis`}. The prefix `Main` means that the constant occurs in head position in the `Conclusion/Hypothesis` of the theorem.
- For each occurrence of a bound variable, we record its *position* in the set {`MainConclusion, MainHypothesis`}. Since bound variables occur almost in every statement, they are not very effective for searching. Thus we are interested only in second order or higher order variables, that are much less frequent. We syntactically recognize a subset of these variables by the fact that they occur in head position: these are the only occurrences we record in the metadata.

[9] E.g. the proof of the fact that 1 is the neutral element of multiplication was given five times. Of course everybody suspects that a proof should already exist. Nevertheless, proving it again is faster than guessing who gave the proof and in what library. As a consequence, the theorem is proved again and the new proof is stored in the library under development by the user, which is likely to be unrelated to arithmetical properties. Thus, without an automatic searching procedure, it is virtually impossible to know that a proof of the theorem was put in that library.

– For each occurrence of a sort *Prop* or *Set*[10], we record its *position* in the set {`MainConclusion, MainHypothesis`}
– For each occurrence in {`MainConclusion, MainHypothesis`} we record its *depth*, i.e. the number of universal quantifiers in the statement or hypothesis.

On the previous metadata we automatically impose both *must constraints* and *only constraints*[11] to achieve both performance and accuracy at the same time[12]. As in the previous cases, our experiments over the Coq library show that the chosen set of constraints is a good compromise between the strictness of the query (in terms of false matches) and the need to get results close to the expected ones (interesting false matches).

Expert users of the system can also edit the set of generated constraints, relaxing both the *must constraints* or the *only constraints*. As a result, they are able to answer similar queries such as: *Retrieve every binary relation* or *Retrieve any n-ary relation over natural numbers*.

Instead of describing here these and other use cases with the introduction of new ad-hoc constraints, we will present in the next section a generic set of constraints that have been identified so far in the projects HELM (Hypertextual Electronic Library of Mathematics, `http://helm.cs.unibo.it`) and MoWGLI (Mathematics on the Web: Get it by Logic and Interfaces, European Project IST-33562, `http://mowgli.cs.unibo.it`). The set of constraints is general enough to subsume the constraints identified in the previous use-cases. Moreover it is relatively stable, since no new constraints have been added in the last few months. The current work is focused on the automatic generation of larger and larger classes of queries based on these constraints. For example, we are now able to write by hand queries to retrieve every theorem in the library of Coq stating the associative or commutative property of any binary operator on any datatype. The number of false matches for these kind of queries is surprisingly low[13]. It is surely possible to generate these queries automatically starting from the formal definition of the wanted property.

5 A Generic Class of Metadata and Constraints

There is an obvious common pattern for the metadata identified in the use cases above: they provide an approximate description of the abstract syntax tree of

[10] *Prop* is the type of every proposition and *Set* is the type of any data-type. They are used to capture second order and higher-order quantifications over properties and types. They can also be used to capture axiom schemas in first-order theories.

[11] With *only constraints* we mean that every occurrence of a constant, bound variable or sort in the matched term must occur in the *only constraints*. In section 2 this test was restricted to the occurrences in the conclusion.

[12] There is no tradeoff here: the *must constraints* can not filter out any good candidate since there are no metavariables in the pattern.

[13] For example, looking for every associative property in the Coq library gives 82 good matches and just 6 false matches.

the statements[14] in the library in terms of the constants and bound variables occurring in them, together with the occurrence locations. This observation leads us to describe our metadata model using *objects* having a list of *references*: the objects represent the statements in the library at the metadata level, i.e. they are the entities the metadata are about, while the references are the metadata describing the occurrences of constants and bound variables in the statements. As we said, an object may have three kinds of references. Namely:

- `refObj` describes the occurrence of a primitive or defined constant in terms of an `occurrence`, a `position` and a `depth` (see below).
- `refSort` describes the occurrence of a sort in terms of a `sort`, a `position` and a `depth` (see below). A sort is the type of propositions or data-types in higher-order logics (i.e. the second order quantification "*for every property* P" can be thought as the typed universal quantification $\forall P : Prop$). Sorts can also be used to capture axiom schemas in first-order theories.
- `refRel` describes the occurrence of a bound variable in terms of a `position` and a `depth` (see below).

The value of a reference is a record made of several fields, whose semantics is:

- `occurrence` specifies the referred defined constant. Its value is an object.
- `sort` specifies the referred sort. Its value belongs to a predefined collection of entities representing the possible sorts at the metadata level.
- `position` specifies the position of the described occurrence in the statement. Its value belongs to a predefined collection of entities representing the possible positions at the metadata level. According to the previous use cases, we identified the following positions:
  - `MainHypothesis`: in head position of a statement premise.
  - `InHypothesis`: in another position of a statement premise.
  - `MainConclusion`: in head position of the statement conclusion.
  - `InConclusion`: in another position of the statement conclusion.
  - `InBody`: not in the statement (for instance in its proof).
- `depth` specifies a depth index associated to the position of an occurrence in the statement. Its value is a natural number. As we saw in the use cases, the `depth` of a reference is defined only when its `position` is set to `MainHypothesis` or to `MainConclusion` and it represents the number of premises of that hypothesis or conclusion.

The above description shows that every reference has some fields for locating the corresponding occurrence in the statement (`position` and `depth`) and may have one field specifying the referred entity (`occurrence` and `sort`)[15].

The queries we generate on these metadata are once again inspired by the use cases we saw in the previous sections. Our general approach concerning query

[14] With statements here we mean the statements of axioms and theorems, and also the types of definitions.

[15] `refRel` does not have this field because, up to now, we did not meet any use case in which we need to discriminate between the occurrences of different bound variables.

generation is that complex queries should be obtained composing basic queries generated from the imposed atomic constraints on the wanted objects.

We identify the following two classes of basic queries:

A. **The wanted objects must have a reference to a given object R (or to a given primitive constant S or to a bound variable) in a given position P with a given depth index D.**
   Here we search for the objects having the assigned values in corresponding fields of at least one of their references. Using the pseudo-language exploited in the previous sections, a query of this class looks like:

```
select every t in the library such that
 the set {
  select every r in t.refObj such that
   r.occurrence = R and r.position = P and r.depth = D
 } is not empty
```

   This class of queries includes what we called the basic *must constraints*.

B. **The wanted objects may have a reference to an object (or to a primitive constant or to a bound variable) only if its position is not included in a given set U of positions, or if it concerns a given object R (or a primitive constant S, or a bound variable) in a given position P with a given depth index D.**
   Here we query the objects such that there are no references whose fields do not have the assigned values. A query of this class may look like:

```
select every t in the library such that
 the set {
  select every r in t.refObj such that
   not (r.occurrence = R and r.position = P and r.depth = D)
   and r.position belongs to U
 } is empty
```

   This formulation exploiting the logical double negation is more convenient, since it can be expressed more easily in SQL. This class of queries includes what we called the basic *only constraints*.

Note that the parameters R, S, P, D, U are always optional.

Coming to the issue of how the above queries should be composed, let us suppose that we are interested in finding the objects satisfying a given set (call it K) of "basic constraints". If the set K contains p constraints in the class A, the wanted objects are meant to satisfy all of them. Since a query of the class A (a *must constraint*) is of the general form

```
M(i) =
 select every t in the library such that
  the set m(i) is not empty
```

the composition of M(1) . . . M(p) is:

```
M = M(1) + ... + M(p) =
 select every t in the library such that
  the set m(1) is not empty and ... and the set m(p) is not empty
```

Dually if the set K contains some constraints in the class B, the wanted objects are meant to satisfy at least one of them. Since a query of the classes B (an *only constraint*) is of the general form:

```
O(j) =
 select every t in the library such that
  the set o(j) is empty
```

the composition of O(1) ... O(q), provided that they concern the same kind of reference, say `refObj`, is:

```
O = O(1) + ... + O(q) =
 select every t in the library such that
  the set o(1) intersected ... intersected o(q) is empty
```

and the same holds for the queries O'(1) ... O'(q') concerning `refSort` and for the queries O"(1) ... O"(q") concerning `refRel`. Now the final query is obtained composing M, O, O' and O" conjunctively:

```
M + O + O' + O'' =
 select every t in the library such that
  the set m(1) is not empty
    and ... and the set m(p) is not empty  and
  the set o(1) intersect ... intersect o(q) is empty  and
  the set o'(1) intersect ... intersect o'(q') is empty  and
  the set o''(1) intersect ... intersect o''(q'') is empty
```

6 Comparison with Other Works

The MBase [14] system resolves a query by matching, by iterating the matching operation over each thesis in its data-base of (which now holds about 6,600 theorems). Thanks to the clever implementation of the unification procedure, the time spent to perform the query is still very low, in the order of 0.8 – 0.9 seconds. Nevertheless, a library of just 6,600 theorems is to be considered still very small; in fact the libraries of the Coq proof-assistant and of the Mizar proof-assistant already hold 40,000 theorems each and they cover just a small subset of the overall mathematical knowledge. This approach will definitely not scale to much larger libraries. To solve the problem, the MBase team is now working on a solution that is extremely close to the one proposed in this paper: the theorems will be indexed in order to avoid unifications that will surely fail. To the authors knowledge, no details on the indexing technology has been published so far.

The Mizar team is now developing a user-friendly query language for the Mizar library [7]. The query language is not based on unification, but it nevertheless quite useful in retrieving interesting results from the library. To retrieve a theorem, the user combines a set of filters over the whole library, until the wanted result is found. The filters are not applied to the library itself, but to a set of indexes (metadata) automatically extracted from the library. Thus the Mizar approach is extremely close to our approach: the main difference is in the set of metadata that, in the case of Mizar, are often very peculiar to the system. For instance, in Mizar it is possible that several symbols share the same notation and there is a query to retrieve the symbols represented by a given notation. Other metadata, instead, are less refined versions of the ones we just presented.

The problem of speeding up matches and unification has already been extensively studied in the literature. In particular, several term indexing techniques have been proposed to speed up matching and unification by storing the whole library in an ad-hoc data-structure particularly optimized for matching. Among the most successful structures, there are discrimination trees, substitution trees and, more recently, coded context trees [11]. The common idea behind these data-structures is to share the term structure as much as possible: two shared substructures are syntactically equal and thus they match one another. Matching and substitutions implemented over these structures can be several orders of magnitude more efficient that iterating unification over the whole library.

Fast term indexing techniques are not incompatible with our approach. In particular, they can be seen as the limit case where the collected metadata are so abundant to describe all the statements in the library and the information about their sharing. Of course, the choice of an efficient data-structure to collect the metadata is a necessary requirement to achieve the gain of fast term indexing. Even if fast term indexing techniques are surely much more performant of our implementation, our general approach has some merits. First of all, since our metadata are RDF triples and not term contexts, they can be easily stored in a common relational data base, while term indexing techniques require specialized libraries implementing complex data-structures. As a consequence, we can easily benefit from the advancements of the data base community, comprising transparent distribution techniques. Moreover, writing spiders to collect new metadata and implementing new kinds of queries is a minor effort in our approach and leaves complete freedom to the implementor. On the contrary, even if the great majority of queries can be reduced to unification, unification is the only operation that is natively provided by term indexing data structures. Moreover, the experience of the Mizar group, also supported by our implementative evidence, shows that unification is often overkilling: the results obtained by querying a very small set of computed metadata are often precise enough to skip the unification step. Finally, the interesting false matches provided by our implementation do provide useful information "for free". To retrieve the same information from term indexing data structures, additional effort must be spent.

7 Conclusions and Future Work

In this paper we presented a general methodology for speeding up queries by matching over large and distributed libraries of mathematical knowledge.

Our approach consists in capturing some of the matching invariants by means of metadata, which can be automatically computed by a batch process (a Web spider). Every time the user submits a pattern to be matched, we use these metadata to filter a small subset of the library items which is made of matching candidates: we compute a query over the metadata from the matching pattern; the results of the query are the matching candidates. The correctness of the filtering operation, which is the property of not loosing matching terms, is a trivial consequence of the fact that our metadata capture the matching invariants.

Depending on the kind of matching we are interested in, we need to identify a different set of metadata and query over them. A second outcome of this work has been to identify a rather reasonable and accurate set of metadata that cover several common matching procedures. They are described in the Section 5.

To judge the efficacy of our approach, we implemented a prototype working over the HELM distributed library, made of about 40.000 theorems coming from the Coq proof-assistant library. The prototype has been implemented using the MathQL Level 1 query language [12] and has been turned into a Web service. A Web interface to the prototype is available following the appropriate link at the address: `http://helm.cs.unibo.it/library.html`. The prototype has been extensively used by one of the authors while developing a medium-sized Coq contribution. The experiment outcome was extremely positive: the author was able to locate and reuse several theorems that were given in parts of the library where the author did not expect them to be. As a consequence, the library reusage factor was improved and the development effort was reduced.

Since the prototype is implemented as a Web service, it can be easily exploited by theorem provers or proof-planners to retrieve lemmas or theorems and thus our search engine may become an important component in the Calculemus perspective. For instance, when a CAS needs to know whether a denominator is never zero, it can require a proof to a proof-planner that can use the search engine to retrieve some existent lemmas used to complete the proof.

An important feature of our filtering technology is that the filtering operation is not too strict, in the sense that wrong candidates that are "close enough" to matching terms are not dropped. These wrong candidates are very interesting, since the human user can often exploit them reducing his thesis to the one found; other times, instead, they can reveal missing assumptions in the user statement.

The interest in the wrong candidates, especially the ones that can be reduced to the user statements, is not a surprise. Indeed, several authors [10, 9] already pointed out that queries by matching are often too strict. For example, we would like the pattern $\forall x.x \geq 0$ to be able to match also the statement $\forall x.0 \leq x$. Other typical examples are patterns involving binary operators: often we would like to match up to associative or commutative rewriting. The most general framework for this kind of query is provided by the so called "query up to isomorphisms" [10]: both the pattern and the statements in the library are reduced to a decidable normal form before the matching; the reduction is such that a proof of the pattern can be automatically produced from the results of the query.

As a future work, we plan to apply our approach in a "query up to isomorphisms" setting. The required modifications are really minimal: Firstly the spider should reduce every term to its normal form *before* extracting the metadata, and secondly the query over the metadata should be generated from the pattern *after* reducing it to normal form. Even in the "query up to isomorphisms" setting, though, we still believe that "almost right" wrong candidates are important to spot false user conjectures and suggest new hypothesis or conditions to be required to make the theorem true.

Other future works involve identifying larger and larger sets of queries that can be automatically generated, possibly enlarging the set of metadata. Applying our technology to other libraries is also interesting to evaluate the scalability of our approach and the level of precision of our metadata. Indeed, the metadata identified in the previous sections are not peculiar of the Coq system, and should be retrieved from any library of mathematical content; however, it may be the case that some libraries could be better indexed with more ad-hoc set of metadatas, as the ones used for Mizar in [7].

References

1. Dewey Decimal Classification
   `http://www.oclc.org/dewey`
2. Library of Congress Classification Scheme,
   `http://www.loc.gov`
3. Mathematical Markup Language (MathML) Version 2.0, W3C Reccomendation 21 February 2001,
   `http://www.w3.org/TR/MathML2`
4. Mathematical Subject Classification, American Mathematical Society,
   `http://www.ams.org/msc`
5. OMDOC: A Standard for Open Mathematical Documents,
   `http://www.mathweb.org/omdoc/omdoc.ps`
6. The OpenMath Standard,
   `http://www.openmath.org/cocoon/openmath/standard/index.html`
7. Grzegorz Bancerek, Piotr Rudnicki, Information Retrieval in MML. In Proceedings of MKM2003. Springler-Verlag LNCS 2594, 2003.
8. Jorge Luis Borges, *The Library of Babel*, in Ficciones, Grove Press 1942.
9. David Delahaye, Roberto di Cosmo, Information Retrieval in a Coq Proof Library using Type Isomorphisms. In Proceedings of TYPES 99, Lökeberg. Springler-Verlag LNCS, 1999.
10. Roberto di Cosmo, *Isomorphisms of Types: from Lambda Calculus to Information Retrieval and Language Design*, Birkhauser, 1995, IBSN-0-8176-3763-X.
11. Harald Ganzinger, Robert Nieuwehuis, Pilar Nivela. Fast Term Indexing with Coded Context Trees. Journal of Automated Reasoning. To appear.
12. Ferruccio Guidi, *Searching and Retrieving in Content-Based Repositories of Formal Mathematical Knowledge*, Ph.D. Thesis in Computer Science, Technical Report UBLCS 2003-06, University of Bologna, March 2003.
13. Conor McBride, *Dependently Typed Functional Programs and their Proofs*, LFCS Ph.D. Thesis, 2000.
14. Michael Kohlhase, Andreas Franke. MBase: Representing Knowledge and Context for the Integration of Mathematical Software Systems, Journal of Symbolic Computation 23:4 (2001), pp. 365–402.

FoCDoc: The Documentation System of FoC

Manuel Maarek[1,2] and Virgile Prevosto[1,3]

[1] L.I.P.6 - Équipe SPI
8, rue du Capitaine Scott - 75015 Paris, France
[manuel.maarek,virgile.prevosto]@spi.lip6.fr
[2] Heriot-Watt University - ULTRA group
Edinburgh EH14 4AS, Scotland UK
manuel.maarek@macs.hw.ac.uk
[3] I.N.R.I.A. - Projet Moscova
B.P. 105 - F-78153 Le Chesnay, France

Abstract. FoC is a computer algebra library with a strong emphasis on formal certification of its algorithms. We present in this article our work on the link between the FoC language and OMDoc, an emerging XML standard to represent and share mathematical contents. On the one hand, we focus on the elaboration of the documentation system FoCDoc. After an analysis of an OMDoc approach of the documentation we present our own XML implementation (FoCDoc) and how we generate, from a FoC program, documentation files in HTML (MathML), LaTeX and OMDoc. On the other hand, we show how it is possible to build tools to translate an OpenMath-OMDoc data into FoC. In other words, we describe in this article how the use of an open standard can help us in providing a friendly interface to the FoC-user, by allowing an interaction between FoC and various rendering engines or editors.

1 Introduction

Thanks to the development of the Internet, more and more documents are available in an electronic form. Moreover, the increasing use of XML [6] allows the author of such documents to express not only presentational constraints, but also to focus on the *semantical content* of the document [10]. This is particularly the case for Computer Algebra Systems (CAS) and Theorem Provers (TPs) with OpenMath [8] and OMDoc [15] which are two emerging XML standards to describe mathematical objects and mathematical structures respectively. The use of such common markup languages offers major benefits in some crucial issues of the design of a mathematical software. In particular:

Documentation: XML offers a wide variety of high-quality publishing tools, either web or paper oriented.

User interaction: If the system is able to handle XML input, then it is possible for the user to use specialized external editors, to enter its data, thus providing a "look and feel" close to the usual mathematical one.

Interaction with other systems: At last, OpenMath and OMDoc can be used to let different CAS and/or TPs interact with each other, by providing them with a common representation of structured data.

In this paper, we show how those three points are addressed within the FoC system. This is done by:

– Translating FoC program contents into OMDoc by strictly enforcing the OMDoc standard format (without adding any FoC-specific tag to it). This translation generates purely formal OMDoc data to ease the communication with other systems. Thus, we share our experience in OMDoc encoding.
– Explaining our implementation choices for the FoCDoc system which generates an OMDoc document and other formats This contribution is a major step in documenting the FoC library and reusing FoC programs.
– Explaining the syntactic and semantic extensions of FoC that allow direct inclusion inside a FoC program of data encoded in the common mathematical format OpenMath-OMDoc.

2 The languages

2.1 The FoC Language

The FoC[4] system provides an environment in which one can write at the same time specifications, programs, and proofs that the programs meet the specifications. We present now the broad outline of the FoC language and express documentation needs for this system. FoC contains two main semantical levels. First, we find *species* and *collections* representing mathematical structures. Second, *functions* and *properties* are the components of *species* and *collections*.

In the FoC language, we use *species* to describe mathematical structures. A *species* is composed of a set of *functions* and *properties*, and has a *carrier type* (introduced by the keyword `rep`) which represents the type of entities that the *species* manipulates. These components are called the *methods* of the species. Each method can be either *declared* or *defined*. A *declared function* is introduced by the keyword `sig` (we only know its signature). A *defined function* is introduced by `let`. A *declared property* (*i.e* a statement) is introduced by `property`, while a *defined* (*i.e.* proved) property deserves the keyword `theorem`. For instance, the species representing groups *declares* an element, `one`, in the carrier and a binary operation `mult`. Then, we have to *declare* the axioms of `one` and `mult` (associativity and neutrality). From that, we can *define* the exponential, and prove some property about it (*e.g* that $x^1 = x$).

A *species* can *inherit* all the components of one or several existing *species*. For instance, a *species* representing abstract rings inherits the *properties* and *functions* of abelian groups and of (multiplicative) monoids.

A *species* can also have *parameters* which can be a *function* –or a constant– or the abstraction of a *collection*. In both cases the parameter's type is specified,

[4] `http://www-spi.lip6.fr/~foc`

using a normal FoC *type* for the first case and a *species*' name in the other case. For instance, a *species* implementing square matrices takes as parameter an integer representing their dimensions, and a *collection* implementing fields to handle the coefficients of the matrices. Parameters differ from inheritance in the sense that a parameter is an abstraction representing *any collection* implementing a given species, while a species inherits only from existing ones.

A *collection* is a completely defined *species*. In other words, every *method* is defined. For instance, we can implement the ring of integers by a *collection* based on the `Gmp` library (a library that manipulates integers of arbitrary size).

2.2 Presentation of OMDoc

Open **M**athematical **Doc**ument [15] is an XML format for encoding mathematical content. It extends OpenMath by adding a *document* and a *theory level* (those extensions decide us to choose OMDoc/OpenMath more than MathML). The *document level* groups all **metadata** informations about the mathematical document. The *theory level* encodes mathematical content. It is structured in **theories** composed of formal (using OpenMath) and informal data. Each *theory* is available as an OpenMath *Content Dictionary.* Moreover, as an open format, OMDoc evolves through multiple experiences, providing good support and reliability.

In the following we explain why we have considered OMDoc as a suitable format for the purposes we have listed previously and why some limitations do not allow to have a pure-OMDoc documentation tool.

3 Designing the FoC Documentation System

3.1 Needs

In our case, needs for a documentation system can be seen from at least two points of view. On the one hand, the programmer wants to have a user friendly interface to navigate trough the language's library as well as in his own FoC program to quickly find and reuse existing FoC `species` and `collections`. On the other hand, the mathematician needs to have a mathematics-oriented output and to easily other mathematical software.

The library. The documentation system should provide facilities to have an overview of already defined `species` and `collections` with their components from FoC and user libraries. This will help the programmer in addition to the already existing tools composing the FoC user interface [19].

Mathematical Knowledge. Perhaps the most important aspect of a documentation system for a CAS is to converse with other mathematical softwares and formats. The next step is to be able to translate existing mathematical format to allow a complete interaction.

Rendering. Since we are dealing with mathematics we need a good rendering of formulae to keep a correspondence with the common mathematical presentation.

3.2 Relations with FoC

We point out the intrinsic links between elements of the FoC language and OMDoc constructions. A summary of those correspondences is given in table 1.

Mathematical structures. FoC's `species` and OMDoc's `<theories>` are quite close. They both represent algebraic structures, are defined by their *components* and may share definitions with other structures by *inheritance* relations.

Components. As said in sec. 2.1, a component of a `species` in FoC can be either a *function* or a *property* or the *carrier type*. It corresponds in OMDoc to a `<symbol>` (elements defined in a theory) or an `<assertion>` (different kinds of statements hold in a theory). *Components* are identified by their names (`<id>` for OMDoc) and the `species` or `<theory>` name in which they appear.

Constructions. Two constructions are important in FoC. First, *inheritance*[5] allows to create a `species` from an existing one. In OMDoc this approach exists also as the `<imports>` directive. It indicates that all symbols and assertions from a given `<theory>` are included in the new `<theory>`. Then the *parametrization* allows to abstract a `species` with respect to a *function* or a *collection*. We explain below why FoC parametrization can not be completely reflected in OMDoc.

Table 1. Correspondences

FoC	OMDoc
`species` and `collection`	`<theory>`
`rep`	`<symbol>`
`sig` and `let`	`<symbol>`
`property` and `theorem`	`<assertion>`
`proof of`	`<proof>`
`inherits` and `implements`	`<imports>`
parameter	`<symbol>`
inheritance with parameters	`<imports>` with `<morphism>`

3.3 Limits

Using only OMDoc as a documentation format suffers two problems. Firstly, we are not able to express correctly some important features of the FoC language. Secondly, we cannot faithfully represent the needs listed in section 3.1.

[5] In FoC, `inherits` (resp. `implements`) construction lists inherited *species* of a new *species* (resp. *collection*).

Expressivity limits. Two kinds of parameters exist in FoC: abstraction of *function* or collection (see section 2.1). This mechanism can be encoded in OMDoc using <imports> and <morphism> which allows to express FoC *function* abstraction.

In the second case, the parameter stand for a collection which should match the interface of a specified species. In OMDoc this kind of parametrization is called **actualization**. Since the parameter is a mathematical structure, it is represented by a <theory>. Table 2 shows an example of the transcription of a parametrized species into OMDoc **actualization**.

Table 2. Parametrization - actualization

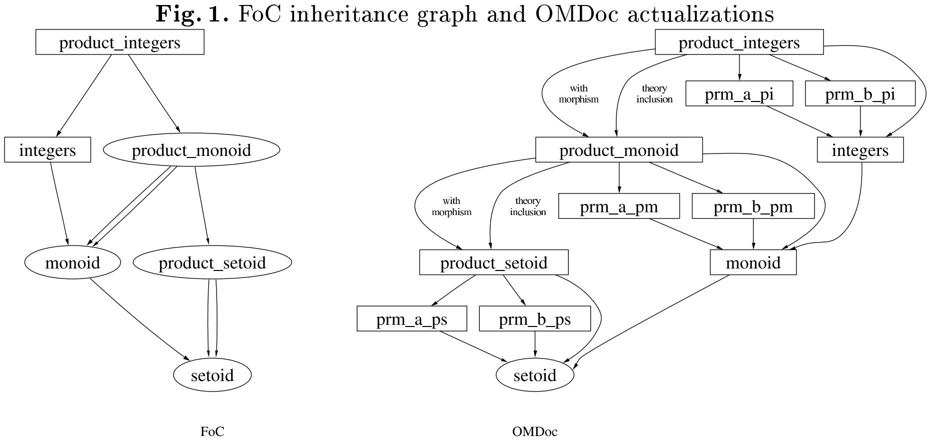
FoC	OMDoc
species fractions (a is gcd_domain) = ... end	<theory id="fractions"> <imports from="gcd_domain"/> </theory>
collection rationals implements fractions(integers) = ... end	<theory id="rationals"> <imports from="integers"/> <imports from="fractions"> <morphism id="gcd-dom-int-morph"> *renaming of each* gcd_domain*'s* *symbol into* integers*' symbols* </morphism></imports></theory>
	<theory-inclusion from="fractions" to="rationals" > <morphism base="gcd-dom-int-morph"/> </theory-inclusion>

Thus, a FoC parameter is represented as an imported <theory> in OMDoc. When a <theory> imports (inherits in the FoC point of view) a parametrized <theory>, one must define a morphism which maps all the symbols of the old parameter to the symbols of a new <theory>. The actualization takes place with this morphism; before the definition of this morphism, there were no information stating that the imported <theory> was playing a parameter role. On the contrary, in FoC, the parameter of a species is directly recognized as a real collection in all the body of the species.

This translation respects the semantics of FoC parameters as well as OMDoc recommendations. The result is a lack of readability and the loss of the FoC structure.

Moreover, if a species has several parameters with the same species as interface, each parameter should be represented by a unique <theory> and each symbol of those parameter <theory> must then be rewritten to avoid confusion when importing them in the target <theory>. Consequently, the following FoC example, in figure 1, would be expressed in OMDoc by a complicated structure.

Fig. 1. FoC inheritance graph and OMDoc actualizations



FoC's *species* are represented by ellipses and *collections* by boxes. FoC's arcs represent *inheritance* and dotted arcs *parametrization*. OMDoc's arcs represent, as in table 2, `<imports>` (with or without `<morphism>`) or `<theory-inclusion>`.

In this example we define the `collection integers`, which implements `monoid` inheriting `setoid`, and `product_integers`, which implements `product_monoid` inheriting `product_setoid`. Each `species product_setoid` and `product_monoid` has two parameters whose interfaces are respectively `setoid` and `monoid`.

Usability limits. It is possible to generate documents of different formats from an OMDoc file. Using XSL transformation files provided in the OMDoc distribution we can get an HTML (including MathML) or a LaTeX file representing a FoC program. This covers the needs of *rendering* and of dealing with the *Mathematical Knowledge* described in section 3.1, but does not provide a good browsing tool for the FoC library. Indeed, the HTML rendering supplied by OMDoc-XSL files is close to the OMDoc format and is guided by the choices of the OMDoc developer team. This output is consequently not linked enough to the FoC grammar and so will not be as helpful as we need for the FoC programmer (for instance, differentiation between `species` and `collection` is no more visible in the generated set of OMDoc **theories**).

4 System Description

Due to the issues mentioned above, our choice has been to build our own XML format, so that it matches all of our needs. From this format we generate HTML and LaTeX files close to FoC syntax as well as OMDoc document.

4.1 The FoCDoc XML Format

The FoCDoc format describes the informations contained in a FoC program that are suitable for documentation. Consequently its DTD (Document Type Definition) is close to the FoC abstract syntax. It contains information coming from three sources.

The FoC concrete syntax. Here, we mean `species`, `collection` and *components* names given by the programmer in the FoC code. We retrieve this information in the FoC abstract syntax.

The type inference and dependencies analysis. The FoC compiler analyzes the correctness of the programs. The type system infers the type of the elements of the language. A second step checks methods dependencies and inheritance resolution. Results of those analysis are stored inside the abstract syntax tree. Using this, we provide all this information in a FoCDoc file:

- the complete list of *components* of *species* and *collections*
- type informations of all elements defined in the program
- the list of ancestors and descendants of *species* and *collections*
- the location of the last definition of a *function* in the inheritance graph
- the location of the proof of an inherited `property` in the inheritance graph

These last two items are very important from a programmer's point of view, since they allow to find very quickly the code of an inherited *function* or the proof of an inherited theorem.

Structured comments. In the FoC concrete syntax, specific comments are analyzed to add informal information in the documentation. Title and authors of the program, common names and comments on elements are concerned here. There are also special commands in the structured comments to specify the encoding and/or rendering of the symbols in TEX, OpenMath, and MathML.

4.2 Machinery

The processing of a document is composed of three steps as described in table 3. We first generate the FoCDoc file using the FoC compiler and its FoCDoc option. Then we apply an XSL transformation (XSLT) to generate a specific XSL file. This file contains a call to a generic XSL and stylesheet templates for symbols rendering. By applying this XSLT to the FoCDoc file we obtain the needed output, OMDoc, HTML or LATEX (depending on which XSL file is used).

5 Interface with Others Systems: Accepting OpenMath/OMDoc Input

As we said in the introduction, the link between FoC and OMDoc is two-sided. Actually, OpenMath objects, and to some extent OMDoc **theories**, are a very

Table 3. Generating process

steps	to obtain a *format* (OMDoc, HTML or LaTeX) output
1	creating FoCDoc file with `focc -focdoc` *file*`.foc` $\longrightarrow$ *file*`.focdoc`
2	applying `focdoc2`*format*`xsl.xsl` *file*`.focdoc` $\longrightarrow$ *file*`.focdoc2`*format*`.xsl`
3	applying *file*`.focdoc2`*format*`.xsl` (which includes `focdoc2`*format*`.xsl`) *file*`.focdoc` $\longrightarrow$ *file*`.`*format*

powerful tool to get information and data from external resources. The main categories of objects that could be given to FoC through an XML representation are the following:

- A mathematical entity, produced by an external editor or another CAS, on which the user wishes to perform some computations within FoC.
- The interface of a `species` that is to be implemented in FoC. As we will see in section 5.2, the fact that there exists an OMDoc model of a given FoC implementation can greatly ease the translation process from an OpenMath entity to its FoC counterpart.
- In the longer run, one could imagine that an OMDoc **theory** may be given to FoC in order to reflect external services that the FoC system could call.

In the remaining of this section, we address the first two points. The third one is described in section 7.

5.1 FoC Extensible Syntax

Before detailing how FoC can be given an XML input, we must introduce one of the major features of the system which is its extensible syntax. Indeed, the parser of the language has been written in Camlp4 [11, 16]. This tool offers the possibility to extend the grammar rules of the FoC syntax without changing the whole system. In other words, it is possible to maintain both a small core syntax for FoC, which eases the analysis of a FoC expression, *and* a lot of syntactic sugar which gives the user the possibility to manipulate expressions close to traditional mathematical syntax. Moreover, since the extensions are not part of the language, it is quite easy to adapt them to a particular "dialect" (For instance, anybody can choose to represent the exponentiation by $**$ or ˆ – or both of course – depending on his own background). The flexibility of the parser allows then the user to write FoC expressions using its own syntax. We focus here on the OpenMath/OMDoc application of this feature.

Besides this extensible syntax, FoC has a datatype `openmath` to represent the main constructions of OpenMath. Let us now see through an example how an OpenMath object is translated into FoC. Suppose that we have implemented $\mathbb{Z}$ in a `collection` called `integers`, and the product $\mathbb{Z} \times \mathbb{Z}$ through a `collection` named `cart`. `cart` itself has a method called `pair` to create an element of `cart` from two elements of `integers`.

By adding a rule[6] which cast any OpenMath integer into an element of `integer`, and another one which binds the other OpenMath constructions into their FoC counterpart, it is possible to let FoC parse the following code:

```
let five=<OMI>5</OMI>;;
let pair =
<OMOBJ><OMA>
  <OMS cd="cart" name="pair"/>
    <OMV name= "#five"/>
    <OMV name= "#five"/>
</OMA></OMOBJ>;;
```

the OMDoc variable `<OMV name="#five">` is used as a reference to the global FoC variable `five` defined just above. The result of the evaluation of `pair` is the product $(5, 5)$ (belonging to `cart`). More generally, there is an easy connexion between OpenMath objects and FoC expressions:

– `<OMA>` are translated into FoC application.
– `<OMBIND>` can be reflected by an abstraction, or a logical quantification, depending on the context where it is used.
– `<OMV>` can refer to a FoC identifier.
– `<OMS>` can be seen as the equivalent of a method call in FoC.

5.2 A FoC to OpenMath lexicon

Even if the correspondence between OpenMath objects and FoC expressions can be very useful to let FoC communicate with other systems, there is still an issue with respect to the names of the OpenMath symbols (`<OMS>`) used in such objects. In fact, the "naive" approach, which binds `<OMS cd="c" name="m">` to the FoC method call `c!m` is very insufficient for two main reasons.

First, there are often several concrete implementations for a given mathematical structure. For instance, multivariate polynomials can be implemented according to a recursive or a distributed representation (see [4]). Depending on the context where they are used, these implementations can be more or less efficient, and a CAS should let the user choose between all of them. On the contrary, OpenMath content dictionaries won't distinguish between those different representations. So we should provide a way to express the bindings between an OpenMath content dictionary and a FoC `collection`. Moreover, these bindings should be accessible from within FoC, so that one can dynamically change the representation which is in use for a given mathematical abstract structure. This can be done easily with an association table, and adding a directive to the FoC syntax such as `change_rep(polynomial,recursive)` to update this table.

[6] Grammar rules are not given here, since it is beyond the scope of this paper to present the whole Camlp4 system.

The second issue concerns the symbol names themselves. Indeed, it might be the case that FoC and OpenMath do not use exactly the same vocabulary for a given structure. For instance, it might be possible that FoC implements the method `add` for a given abelian group, while OpenMath expects **plus** for the same structure. We could use another association table to avoid that, but, given that a mathematical structure can be quite complex, such a table might be difficult to create and to maintain. It seems much more profitable to take advantage of the OMDoc standard to formalize the symbols that are recognized by a FoC library. To achieve that, one can take an existing OMDoc **theory** $\mathcal{T}$, and derive a FoC `species` t from it. An important precondition for that would be that the formal description of every symbol of $\mathcal{T}$ includes its type. For instance, a minimal **theory** to describe polynomials would include the following symbols: (types are omitted due to space constraints).

```
<theory id="poly">
  <symbol id="add"> ... </symbol>
  <symbol id="monomial"> ... </symbol>
  <symbol id="constant"> ... </symbol>
  <symbol id="mult"> ... </symbol>
</theory>
```

It can be imported as a completely abstract `species poly` in FoC. Then, we can derive one or more `collection` implementing `poly` through the normal inheritance process of FoC. Under the assumption that such an implementation exists and is called `my_poly`, then FoC will be able to parse the program

```
change_rep(poly, my_poly);;
let p =
<OMOBJ><OMA>
  <OMS cd="poly" name="add"/>
  <OMA><OMS cd="poly" name="monomial"/><OMSTR>X</OMSTR><OMI>2</OMI></OMA>
  <OMA><OMS cd="poly" name="constant"/><OMI>5</OMI></OMA>
</OMA></OMOBJ>;;
```

so that the variable `p` is bound to the representation of the polynomial $X^2 + 5$ in `my_poly`. Indeed, since `my_poly` is an implementation of `poly`, it must provide a definition for all the methods of `poly`, that is all the symbols of the OMDoc **theory** `<poly>`. Then, it is sufficient to verify that the XML code refers to existing symbols to be guaranteed that it will be accepted by FoC.

6 Related Work

A certain number of projects are using XML as an exchange or publishing format. Among them, we should cite the HELM project [1], which proposes its own DTD, oriented toward TPs. In particular, HELM's online library [2], allows one to browse through a huge amount of mathematical theorems. Even if HELM's DTD might not be completely suited for a CAS like FoC, it might be of a great interest when considering the certification part of FoC, as a possible interface with other TPs. One of the main attempts to integrate different

CAS and TPs inside a common framework has been done inside the MathWeb project[7]. MathWeb is distributed through different locations (Edinburgh, Pittsburgh, Saarbrücken, ...), where different services are interacting with each other along the MathWeb Software Bus [21]. The communication between the various services, which include both CAS and TPs is done through OMDoc objects.

7 Perspectives

FoCDoc improvements We see two important issues in the FoCDoc's future. Firstly, after having improved the fully commented FoC library we should be able to elaborate a searching engine for it such as the one developed for the Mizar Mathematical Library [3]. Secondly, we should increase the usability of generated documents by linking them with the other tools of the user interface (listed in [19]).

Toward an integration in the MathWeb software bus? In the longer run, the integration of FoC inside MathWeb seems to be a very promising feature. On the one hand, we could of course use FoCDoc to produce OMDoc **theories** to describe the capabilities of the FoC library, so that other systems could make requests to the FoC system as OpenMath objects.

On the other hand, one could imagine that FoC itself might send requests on the software bus, according to the theories that other systems have exposed and are able to handle. Since there are not only CAS but also TPs in MathWeb, such an interaction is particularly interesting for the certification part of the FoC project, which could then be able to delegate some proofs to some state-of-the-art TPs.

A Front-end for FoC Last, the use of OpenMath/OMDoc as an input language for FoC offers the possibility to use some graphical mathematical editors which have the possibility to generate OpenMath content. Such editors might be a convenient way for the end-user to do some computations in FoC without having to learn all the syntax, together with the list of `collections` and methods available.

8 Conclusion

As a conclusion, we can say that the use of an open XML standard both as an input and as an output language for FoC is a very convincing experience. First, it allows us to document fairly easily the FoC library, which is a crucial point of its development, for a great variety of media. Second, it eases the interaction between FoC and other systems. Thanks to that, we can concentrate our efforts on the core of the project, that is the development of the FoC compiler and of the library, while offering to the end-user a decent interface.

[7] `http://www.mathweb.org`

References

1. Andrea Asperti, Luca Padovani, Claudio Sacerdoti Coen, and Irene Schena. Helm and the semantic math-web. In *Proceedings of TPHOLs*, LNCS, 2001.
2. Andrea Asperti, Irene Schena, Luca Padovani, Ferruccio Guidi, Claudio Sacerdoti Coen, and Stefano Zacchiroli. HEΛM, An Hypertextual Electronic Library of Mathematics. `http://le.cs.unibo.it/helm`.
3. Grzegorz Bancerek and Piotr Rudnicki. Information retrieval in mml. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *Proceedings of MKM'03*. Springer, 2003.
4. Sylvain Boulmé, Thérèse Hardin, and Renaud Rioboo. Some hints for polynomials in the Foc project. In *Calculemus 2001 Proceedings*, June 2001.
5. S. Boulmé, T. Hardin, D. Hirschkoff, V. Ménissier-Morain, and R. Rioboo. On the way to certify computer algebra systems. In *Proceedings of the Calculemus workshop of FLOC'99*, volume 23 of *ENTCS*. Elsevier, 1999.
6. Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0 (second edition). W3c recommendation, World Wide Web Consortium, October 2002. `http://www.w3.org/TR/REC-xml`.
7. B. Buchberger et al. A survey on the theorema project. In W. Kuechlin, editor, *Proceedings of ISSAC'97*. ACM Press, 1997.
8. O. Caprotti, D. P. Carlisle, and A. M. Cohen. *The OpenMath Standard*. The OpenMath Esprit Consortium, `http://www.openmath.org`, February 2000.
9. Olga Caprotti, Herman Geuvers, and Martijn Oostdijk. Certified and portable mathematical documents from formal contexts. In *Proceedings of MKM 2001*, Linz, Austria, September 2001.
10. World Wide Web Consortium. Semantic web. `http://www.w3.org/2001/sw/`.
11. Daniel de Rauglaudre. *Camlp4 Reference Manual – version 3.06*. INRIA, 2002. available at `http://caml.inria.fr/camlp4/manual/`.
12. Stephen Deach. Extensible stylesheet language (xsl) recommendation. W3c recommendation, World Wide Web Consortium, November 1999. `http://www.w3.org/TR/xslt`.
13. Andreas Franke and Michael Kohlhase. System Description: MathWeb, an Agent-Based Communication Layer for Distributed Automated Theorem Proving. In *Conference on Automated Deduction*, pages 217–221, 1999.
14. Andreas Franke and Michael Kohlhase. System description: MBASE, an open mathematical knowledge base. In *Conference on Automated Deduction*, 2000.
15. Michael Kohlhase. *OMDoc: An Open Markup Format for Mathematical Documents (Version 1.1)*. `http://www.mathweb.org/omdoc`, 2003.
16. Manuel Maarek. Écriture d'un parseur pour FOC en Camlp4. Travail d'Initiation à la Recherche, Université Paris 6, June 2001. In French.
17. Manuel Maarek. Conception d'une librairie omdoc pour foc. Rapport de stage de D.E.A, Université Paris 6, September 2002. In French.
18. Virgile Prevosto and Damien Doligez. Algorithms and proof inheritance in the FoC language. *Journal of Automated Reasoning*, 29(3-4):337–363, December 2002.
19. FoC Project. *The FoC System Reference Manual Version 0*. SPI LIP6, 2003.
20. J. Zimmer and L. Dennis. Inductive Theorem Proving and Computer Algebra in the MathWeb Software Bus. In *Proceedings of the 10th CALCULEMUS Symposium 2002*, Marseille (France), July 2002.
21. Jürgen Zimmer and Michael Kohlhase. System description: The mathweb software bus for distributed mathematical reasoning. In *Proceedings of the 18th International Conference on Automated Deduction*, LNAI, 2002.

Brokers and Web-Services for Automatic Deduction: a Case Study

Claudio Sacerdoti Coen* and Stefano Zacchiroli**

[1] Department of Computer Science
University of Bologna
Mura Anteo Zamboni 7, 40127 Bologna, ITALY
sacerdot@cs.unibo.it
[2] Department of Computer Science
École Normale Supérieure
45, Rue d'Ulm, F-75230 Paris Cedex 05, FRANCE
zack@cs.unibo.it

Abstract. We present a planning broker and several Web-Services for automatic deduction. Each Web-Service implements one of the tactics usually available in interactive proof-assistants. When the broker is submitted a "proof status" (an incomplete proof tree and a focus on an open goal) it dispatches the proof to the Web-Services, collects the successful results, and send them back to the client as "hints" as soon as they are available.
In our experience this architecture turns out to be helpful both for experienced users (who can take benefit of distributing heavy computations) and beginners (who can learn from it).

1 Introduction

The Web-Service approach at software development seems to be a working solution for getting rid of a wide range of incompatibilities between communicating software applications. W3C's efforts in standardizing related technologies grant longevity and implementations availability for frameworks based on Web-Services for information exchange. As a direct consequence, the number of such frameworks is increasing and the World Wide Web is moving from a disorganized repository of human-understandable HTML documents to a disorganized repository of applications working on machine-understandable XML documents both for input and output.

The big challenge for the next future is to provide stable and reliable services over this disorganized, unreliable, and ever-evolving architecture. The standard solution is to provide a further level of stable services (called *brokers*) that behave

* Partially supported by 'MoWGLI: Math on the Web, Get it by Logic and Interfaces', EU IST-2001-33562

** Partially supported by 'MyThS: Models and Types for Security in Mobile Distributed Systems', EU FET-GC IST-2001-32617

as common gateways/addresses for client applications to access a wide variety of services and abstract over them.

Since the *Declaration of Linz*, the MONET Consortium[3] is working on the development of a framework, based on the Web-Services/brokers approach, aimed at providing a set of software tools for the advertisement and the discovery of mathematical Web-Services.

Several groups have already developed software bus and services[4] providing both computational and reasoning capabilities [3, 4, 15, 16]: the first ones are implemented on top of Computer Algebra Systems; the second ones provide interfaces to well-known theorem provers. Proof-planners, proof-assistants, CASs and domain-specific problem solvers are natural candidates to be clients of these services. Nevertheless, so far the number of examples in the literature has been insufficient to fully assess the concrete benefits of the framework.

In this paper we present an architecture, namely H-Bugs, implementing a *suggestion engine* for the proof assistant developed on behalf of the HELM[5] project [5]. We provide several Web-Services (called *tutors*) able to suggest possible ways to proceed in a proof. The tutors are orchestrated by a broker (a Web-Service itself) that is able to dispatch a proof status from a client (the proof-assistant) to the tutors; each tutor tries to make progress in the proof and, in case of success, notifies the client that shows an *hint* to the user. The broker is an instance of the homonymous entity of the MONET framework. The tutors are MONET services. Another Web-Service (which is not described in this paper and which is called Getter [14]) is used to locate and download mathematical entities; the Getter plays the role of the Mathematical Object Manager of the MONET framework.

A precursor of H-Bugs is the Ωmega-Ants project [6, 7], which provided similar functionalities to the Ωmega proof-planner [8]. The main architectural difference between H-Bugs and Ωmega-Ants is that the latter is based on a blackboard architecture and it is not implemented using Web-Services and brokers.

In Sect. 2 we present the architecture of H-Bugs. A usage session is shown in Sect. 3. Further implementation details are given in Sect. 4. Sect. 5 is an overview of the tutors that have been implemented. As usual, the final section of this paper is devoted to conclusions and future works.

2 An H-Bugs Bird's Eye View

The H-Bugs architecture (depicted in Fig. 1) is based on three different kinds of actors: *clients*, *brokers*, and *tutors*. Each actor presents one or more Web-Service interfaces to its neighbors H-Bugs actors.

[3] `http://monet.nag.co.uk/cocoon/monet/index.html`

[4] The most part of these systems predate the development of Web-Services. Those systems whose development is still active are slowly being reimplemented as Web-Services.

[5] Hypertextual Electronic Library of Mathematics, `http://helm.cs.unibo.it`

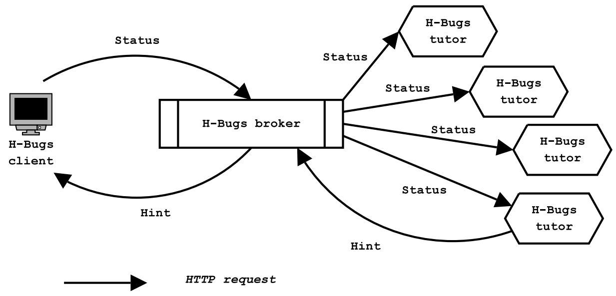


Fig. 1. H-Bugs architecture

In this section we detail the role and requirements of each kind of actors and we discuss about the correspondences between them and the MONET entities described in [11]. Due to lack of space, we cannot compare our framework to similar proposals, as the older and more advanced Ωmega system. The study of the correspondences with MONET is well motivated by the fact that the MONET framework is still under development and that our implementation is one of the first experiments in Web-Servicebased distributed reasoning. On the other hand, a comparison with Ωmega would be less interesting since the functionalities we provide so far are just a subset of the Ωmega-Ants ones.

Clients An H-Bugs client is a software component able to produce *proof status* and to consume *hints*.

A proof status is a representation of an incomplete proof and is supposed to be informative enough to be used by an interactive proof assistant. No additional requirements exist on the proof status, but there should be an agreement on its format between clients and tutors. A hint is an encoding of a step that can be performed in order to proceed in an incomplete proof. Usually it represents a reference to a tactic available on some proof assistant along with an instantiation for its formal parameters. Hints can also be more structured: a hint can be as complex as a whole proof-plan.

Using W3C's terminology [1], clients act both as Web-Service providers and requesters, see Fig. 2. They act as providers receiving hints from the broker; they act as requesters submitting new status to the tutors. Clients additionally use broker services to know which tutors are available and to subscribe to one or more of them.

Usually, when the client role is taken by an interactive proof assistant, new status are sent to the broker as soon as the proof change (e.g. when the user applies a tactic or when a new proof is started); hints are shown to the user by the means of some effects in the user interface (e.g. popping a dialog box or enlightening a tactic button).

H-Bugs clients act as MONET clients and ask brokers to provide access to a set of services (the tutors). H-Bugs has no actors corresponding to MONET's

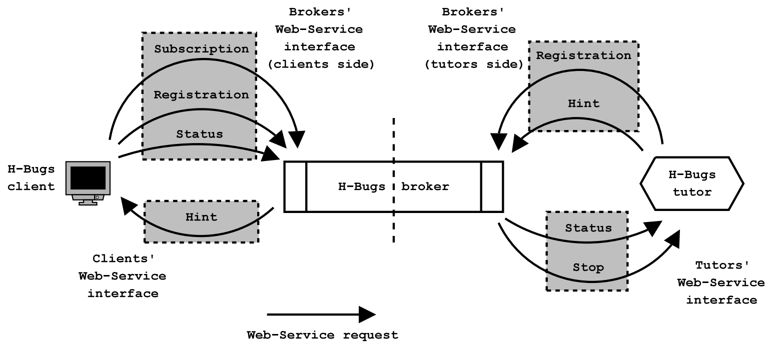


Fig. 2. H-Bugs Web-Services interfaces

Broker Locating Service (since the client is supposed to know the URI of at least one broker). The H-Bugs clients and tutors contact the Getter (a MONET Mathematical Object Manager) to locate and retrieve mathematical items from the HELM library. The proof status that are exchanged by the H-Bugs actors, instead, are built on the fly and are neither stored nor given an unique identifier (URI) to be managed by the Getter.

Brokers Brokers are the key actors of the H-Bugs architecture since they act as intermediaries between clients and tutors. They behave as Web-Services providers and requesters for *both* clients and tutors, see Fig. 2.

With respect to the client, a broker acts as a Web-Service provider, receiving the proof status and forwarding it to one or more tutors. It also acts as a Web-Service requester sending hints to the client as soon as they are available from the tutors.

With respect to the tutors, the Web-Service provider role is accomplished by receiving hints as soon as they are produced; as a requester, it is accomplished by asking for computations (*musings* in H-Bugs terminology) on status received by clients and by stopping already late but still ongoing `musings`.

Additionally brokers keep track of available tutors and clients subscriptions.

H-Bugs brokers act as MONET brokers implementing the following components: Client Manager, Service Registry Manager (keeping track of available tutors), Planning Manager (choosing the available tutors among the ones to which the client is subscribed), Execution Manager. The Service Manager component is not required since the session handler, that identifies a session between a service and a broker, is provided to the service by the broker instead of being received from the service when the session is initialized. In particular, a session is identified by an unique identifier for the client (its URL) and an unique identifier for the broker (its URL).

Notice that H-Bugs brokers have no knowledge of the domain area of proof-assistants, nor they are able to interpret the messages that they are forwarding.

They are indeed only in charge of maintaining the abstraction of several reasoning blackboards — one for each client — of capacity one: a blackboard is created when the client submits a problem; it is then "shared" by the client and all the tutors until the client submits the next problem. For instance, replacing the client with a CAS and all the tutors with agents implementing different resolution methods for differential equations would not require any change in the broker. Notice that all the tutors must expose the same interface to the broker.

The MONET architecture specification does not state explicitly whether the service and broker answers can be asynchronous. Nevertheless, the described information flow implicitly suggests a synchronous implementation. On the contrary, in H-Bugs every request is asynchronous: the connection used by an actor to issue a query is immediately closed; when a service produces an answer, it gives it back to the issuer by calling the appropriate actor's method.

Tutors Tutors are software components able to consume proof status producing hints. H-Bugs does not specify by which means hints should be produced: tutors can use any means necessary (heuristics, external theorem prover or CAS, etc.). The only requirement is that there exists an agreement on the formats of proof status and hints.

Tutors act both as Web-Service providers and requesters for the broker, see Fig. 2. As providers, they wait for commands requesting to start a new `musing` on a given proof status or to stop an old, out of date, `musing`. As requesters, they signal to the broker the end of a `musing` along with its outcome (a hint in case of success or a failure notification).

H-Bugs tutors act as MONET services.

3 An H-Bugs Session Example

In this section we describe a typical H-Bugs session. The aim of the session is to solve the following easy exercise:

Exercise 1. Let x be a generic real number. Using the HELM proof-engine, prove that

$$x = \frac{(x+1)*(x+1)-1-x*x}{2}$$

Let us suppose that the H-Bugs broker is already running and that the tutors already registered themselves to the broker. When the user starts our proof-engine `gTopLevel`, the system registers itself to the broker, that sends back the list of available tutors. By default, `gTopLevel` notifies to the broker its intention of subscribing to every tutor available. The user can always open a configuration window where she is presented the list of available tutors and she can independently subscribe and unsubscribe herself to each tutor.

The user can now insert into the system the statement of the theorem and start proving it. Let us suppose that the first step of the user is proving that the denominator 2 is different from 0. Once that this technical result is proven,

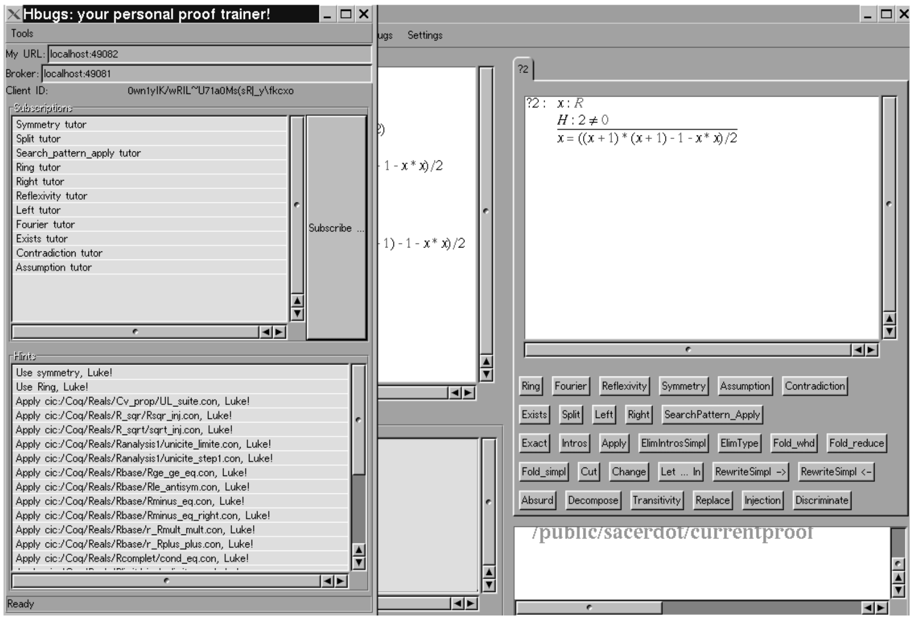


Fig. 3. Example session.

the user must prove the goal shown in the upper right corner of the window in background in Fig. 3.

While the user is wondering how to proceed in the proof, the tutors are trying to progress in the proof. After a while, the tutors' suggestions start to appear in the lower part of the H-Bugs interface window (the topmost window in Fig. 3). In this case, the tutors are able to produce 23 hints. The first and not very useful hint suggests to proceed in the proof by exchanging the two sides of the equality. The second hint suggests to reduce both sides of the equality to their normal form by using only reductions which are justified by the ring structure of the real numbers; the two normal forms, though, are so different that the proof is not really simplified. All the residual 21 hints suggest to apply one lemma from the distributed library of HELM. The user can look at the statement of any lemma by clicking on its URI.

The user can now look at the list of suggestions and realize that a good one is applying the lemma `r_Rmult_mult` that allows to multiply both equality members by the same scalar[6]. Double-clicking on the hint automatically applies the lemma, reducing the proof to closing three new goals. The first one asks the user the scalar to use as an argument of the previous lemma; the second one

[6] Even if she does not receive the hint, the user probably already knows that this is the right way to proceed. The difficult part, accomplished by the hint, is guessing what is the name of the lemma to apply.

states that the scalar is different from 0; the third lemma (the main one) asks to prove the equality between the two new members.

The user proceeds by instantiating the scalar with the number 2. The `Assumption` tutor now suggests to close the second goal (that states that $2 \neq 0$) by applying the hypothesis H. No useful suggestions, instead, are generated for the main goal $2*x = 2*((x+1)*(x+1)-1-x*x)*2^{-1}$. To proceed in the proof the user needs to simplify the expression using the lemma *Rinv_r_simpl_m* that states that $\forall x, y.\, y = x*y*x^{-1}$. Since we do not provide yet any tutor suggesting simplifications, the user must find out this simplification by himself. Once she founds it, the goal is reduced to proving that $2*x = (x+1)*(x+1)-1-x*x$. This equality is easily solved by the `Ring` tutor, that suggests[7] to the user how to complete the proof in one macrostep.

4 Implementation's Highlights

In this section we present some of the most relevant implementation details of the H-Bugs architecture.

Proof status In our implementation of the H-Bugs architecture we used the proof assistant of the HELM project (codename `gTopLevel`) as an H-Bugs client. Thus we have implemented serialization/deserialization capabilities for its internal status. In order to be able to describe Web-Services that exchange status in WSDL using the XML Schema type system, we have chosen an XML format as the target format for the serialization.

Each proof is represented by a tuple of four elements: *uri*, *metasenv*, *proof*, *thesis*.

uri : an URI chosen by the user at the beginning of the proof process. Once (and if) proved, that URI will globally identify the term inside the HELM library (given that the user decides to save it).

thesis : the ongoing proof thesis

proof : the current incomplete proof tree. It can contain *metavariables* (holes) that stands for the parts of the proof that are still to be completed. Each metavariable appearing in the tree references one element of the metavariables environment (*metasenv*).

metasenv : the metavariables environment is a list of *goals* (unproved conjectures). In order to complete the proof, the user has to instantiate every metavariable in the proof with a proof of the corresponding goal. Each goal is identified by an unique identifier and has a context and a type (the goal thesis). The context is a list of named hypotheses (declarations and definitions). Thus the context and the goal thesis form a sequent, which is the statement of the proof that will be used to instantiate the metavariable occurrences.

[7] The `Ring` suggestion is just one of the 22 hints that the user receives. It is the only hint that does not open new goals, but the user right now does not have any way to know that.

Each of these information is represented in XML as described in [12]. Additionally, an H-Bugs status carries the unique identifier of the current goal, which is the goal the user is currently focused on. Using this value it is possible to implement different client side strategies: the user could ask the tutors to work on the goal she is considering or to work on the other "background" goals.

Hints A hint in the H-Bugs architecture should carry enough information to permit the client to progress in the current proof. In our implementation each hint corresponds to either one of the tactics available to the user in gTopLevel (together with its actual arguments) or a set of alternative suggestions (a list of hints).

For tactics that do not require any particular argument (like tactics that apply type constructors or decision procedures) only the tactic name is represented in the hint. For tactics that need terms as arguments (for example the `Apply` tactic that apply a given lemma) the hint includes a textual representation of them, using the same representation used by the interactive proof assistant when querying user for terms. In order to be transmitted between Web-Services, hints are serialized in XML.

It is also possible for a tutor to return more hints at once, grouping them in a particular XML element. This feature turns out to be particularly useful for the *searchPatternApply* tutor (see Sect. 5) that queries a lemma database and returns to the client a list of all lemmas that could be used to complete the proof. This particular hint is encoded as a list of `Apply` hints, each of them having one of the results as term argument.

We would like to stress that the H-Bugs architecture has no dependency on either the hint or the status representation: the only message parts that are fixed are those representing the administrative messages (the envelopes in the Web-Services terminology). In particular, the broker can manage at the same time several sessions working on different status/hints formats. Of course, there must be an agreement between the clients and the tutors on the format of the data exchanged.

In our implementation the client does not trust the tutors hints: being encoded as references to available tactics imply that an H-Bugs client, at the receipt of a hint, simply try to replay the work done by a tutor on the local copy of the proof. The application of the hint can even fail to type check and the client copy of the proof can be left undamaged after spotting the error. Note, however, that it is still possible to implement a complex tutor that looks for a proof doing backtracking and that send back to the client a hint whose argument is a witness (a trace) of the proof found: the client applies the hint reconstructing (and checking the correctness of) the proof from the witness, without having to re-discover the proof itself.

An alternative implementation where the tutors are trusted would simply send back to the client a new proof-status. Upon receiving the proof-status, the client would just override its current proof status with the suggested one. In the case of those clients which are implemented using proof-objects (as the Coq proof-assistant, for instance), it is still possible for the client to type-check the proof-object and reject wrong hints. The systems that are not based on proof-objects (as PVS, NuPRL, etc.), instead, must completely trust the new proof-status. In this case the H-Bugs architecture would need at least to be extended with clients-tutors authentication.

Registries Being central in the H-Bugs architecture, the broker is also responsible of housekeeping operations both for clients and tutors. These operations are implemented using three different data structures called *registries*: clients registry, tutors registry and `musings` registry.

In order to use the suggestion engine a client should register itself to the broker and subscribe to one or more tutors. The registration phase is triggered by the client using the `Register_client` method of the broker to send him an unique identifier and its base URI as a Web-Service. After the registration, the client can use the `List_tutors` method of the broker to get a list of available tutors. Eventually the client can subscribe to one or more of these using the `Subscribe` method of the broker. Clients can also unregister from brokers using `Unregister_client` method.

The broker keeps track of both registered clients and clients' subscriptions in the clients registry.

In order to be advertised to clients during the subscription phase, tutors should register to the broker using the `Register_tutor` method of the broker. This method is really similar to `Register_client`: tutors are required to send an unique identifier and a base URI for their Web-Service. Additionally tutors are required to send an human readable description of their capabilities; this information could be used by the client user to decide which tutors she wants to subscribe to. As the clients, tutors can unregister from brokers using `Unregister_broker` method.

Each time the client status changes, it get sent sent to the broker using its `Status` method. Using both the clients registry (to lookup the client's subscription) and the tutors registry (to check if some tutors have unsubscribed), the broker is able to decide to which tutors the new status have to be forwarded.

The forwarding operation is performed using the `Start_musing` method of the tutors, that is a request to start a new computation (*`musing`*) on a given status. The return value of `Start_musing` is a `musing` identifier that is saved in the `musings` registry along with the identifier of the client that triggered the `musing`.

As soon as a tutor completes an `musing`, it informs the broker using its `Musing_completed` method; the broker can now remove the `musing` entry from the `musings` registry and, depending on its outcome, inform the client. In case of success one of the `Musing_completed` arguments is a hint to be sent to the client; otherwise there is no need to inform him and the `Musing_completed` method is called just to update the `musings` registry.

Consulting the `musings` registry, the broker is able to know, at each time, which `musings` are in execution on which tutor. This peculiarity is exploited by the broker on invocation of the `Status` method. Receiving a new status from the client implies indeed that the previous status no longer exists and all `musings` working on it should be stopped: additionally to the already described behavior (i.e. starting new `musings` on the received status), the broker takes also care of stopping ongoing computation invoking the `Stop_musing` method of the tutors.

Tutors Each tutor exposes a Web-Service interface and should be able to work, not only for many different clients referring to a common broker, but also for many different brokers. The potential high number of concurrent clients imposes a multi-threaded or multi-process architecture.

Our current implementation is based on a multi threaded architecture exploiting the capabilities of the O'HTTP library [14]. Each tutor is composed by one always running thread plus an additional thread for each `musing`. One thread is devoted to listening for incoming Web-Service requests; when a request is received the control is passed to a second thread, created on the fly, that handle the incoming request (usual one-thread-per-request approach in web servers design). In particular if the received request is `Start_musing`, a new thread is spawned to handle it; the thread in duty to handle the HTTP request returns an HTTP response containing the identifier of the just started `musing`, and then dies. If the received request is `Stop_musing`, instead, the spawned thread kills the thread responsible for the `musing` whose identifier is the argument of the `Stop_musing` method.

This architecture turns out to be scalable and allows the running threads to share the cache of loaded (and type-checked) theorems. As we will explain in Sect. 5, this feature turns out to be really useful for tactics that rely on a huge but fixed set of lemmas, as every reflexive tactic.

The implementation of a tutor within the described architecture is not that difficult having a language with good threading capabilities (as OCaml has) and a pool of already implemented tactics (as `gTopLevel` has). Working with threads is known to be really error prone due to concurrent programming intrinsic complexity. Moreover, there is a non-neglectable part of code that needs to be duplicated in every tutor: the code to register the tutor to the broker and to handle HTTP requests; the code to manage the creation and termination of threads; and the code for parsing the requests and serializing the answers. As a consequence we have written a generic implementation of a tutor which is parameterized over the code that actually proposes the hint and over some administrative data (as the port the tutor will be listening to).

The generic tutor skeleton is really helpful in writing new tutors. Nevertheless, the code obtained by converting existing tactics into tutors is still quite repetitive: every tutor that wraps a tactic has to instantiate its own copy of the proof-engine kernel and, for each request, it has to override its status, guess the tactic arguments, apply the tactic and, in case of success, send back a hint with the tactic name and the chosen arguments. Of course, the complex part of the work is guessing the right arguments. For the simple case of tactics that do not require any argument, though, we are able to automatically generate the whole tutor code given the tactic name. Concretely, we have written a tactic-based tutor template and a script that parses an XML file with the specification of the tutor and generates the tutor's code. The XML file describes the tutor's port, the code to invoke the tactic, the hint that is sent back upon successful application and a human readable explanation of the tactic implemented by the tutor.

5 The Implemented H-BugsTutors

To test the H-Bugs architecture and to assess the utility of a suggestion engine for the end user, we have implemented several tutors. In particular, we have investigated three classes of tutors:

1. *Tutors for beginners.* These are tutors that implement tactics which are neither computationally expensive nor difficult to understand: an expert user can always understand if the tactic can be applied or not without having to try it. For example, the following implemented tutors belong to this class:
   - *Assumption Tutor*: it ends the proof if the thesis is equivalent[8] to one of the hypotheses[9].
   - *Contradiction Tutor*: it ends the proof by *reductio ad adsurdum* if one hypothesis is equivalent to $False$.
   - *Symmetry Tutor*: if the goal thesis is an equality, it suggests to apply the commutative property.
   - *Left/Right/Exists/Split/Reflexivity/Constructor Tutors*: the Constructor Tutor suggests to proceed in the proof by applying one or more constructors when the goal thesis is an inductive type or a proposition inductively defined according to the declarative style[10]. Since disjunction, conjunction, existential quantification and Leibniz equality are particular cases of inductive propositions, all the other tutors of this class are instantiations of the the Constructor tactic. Left and Right suggest to prove a disjunction by proving its left/right member; Split reduces the proof of a conjunction to the two proof of its members; Exists suggests to prove an existential quantification by providing a witness[11]; Reflexivity proves an equality whenever the two sides are convertible.

[8] In our implementation, the equivalence relation imposed by the logical framework is *convertibility*. Two expressions are convertible when they reduce to the same normal form. Two "equal" terms depending on free variables can be non-convertible since free variables stop the reduction. For example, $2x$ is convertible with $(3-1)x$ because they both reduce to the same normal form $x+x+0$; but $2x$ is not convertible to $x2$ since the latter is already in normal form.

[9] In some cases, especially when non-trivial computations are involved, the user is totally unable to figure out the convertibility of two terms. In these cases the tutor becomes handy also for expert users.

[10] An example of a proposition that can be given in declarative style is the $\leq$ relation over natural numbers: $\leq$ is the smallest relation such that $n \leq n$ for every n and $n \leq m$ for every n, m such that $n \leq p$ where p is the predecessor of m. Thus, a proof of $n \leq n$ is simply the application of the first constructor to n and a proof of $n \leq m$ is the application of the second constructor to n, m and a proof of $n \leq m$.

[11] This task is left to the user.

Beginners, when first faced with a tactic-based proof-assistant, get lost quite soon since the set of tactics is large and their names and semantics must be remembered by heart. Tutorials are provided to guide the user step-by-step in a few proofs, suggesting the tactics that must be used. We believe that our beginners tutors can provide an auxiliary learning tool: after the tutorial, the user is not suddenly left alone with the system, but she can experiment with variations of the exercises given in the tutorial as much as she like, still getting useful suggestions. Thus the user is allowed to focus on learning how to do a formal proof instead of wasting efforts trying to remember the interface to the system.

2. *Tutors for Computationally Expensive Tactics.* Several tactics have an unpredictable behavior, in the sense that it is unfeasible to understand whether they will succeed or they will fail when applied and what will be their result. Among them, there are several tactics either computationally expensive or resource consuming. In the first case, the user is not willing to try a tactic and wait for a long time just to understand its outcome: she would prefer to keep on concentrating on the proof and have the tactic applied in background and receive out-of-band notification of its success. The second case is similar, but the tactic application must be performed on a remote machine to avoid overloading the user host with several concurrent resource consuming applications.
   Finally, several complex tactics and in particular all the tactics based on reflexive techniques depend on a pretty large set of definitions, lemmas and theorems. When these tactics are applied, the system needs to retrieve and load all the lemmas. Pre-loading all the material needed by every tactic can quickly lead to long initialization times and to large memory footstamps. A specialized tutor running on a remote machine, instead, can easily pre-load the required theorems.
   As an example of computationally expensive task, we have implemented a tutor for the *Ring* tactic [9]. The tutor is able to prove an equality over a ring by reducing both members to a common normal form. The reduction, which may require some time in complex cases, is based on the usual commutative, associative and neutral element properties of a ring. The tactic is implemented using a reflexive technique, which means that the reduction trace is not stored in the proof-object itself: the type-checker is able to perform the reduction on-the-fly thanks to the conversion rules of the system. As a consequence, in the library there must be stored both the algorithm used for the reduction and the proof of correctness of the algorithm, based on the ring axioms. This big proof and all of its lemmas must be retrieved and loaded in order to apply the tactic. The Ring tutor loads and caches all the required theorems the first time it is contacted.
3. *Intelligent Tutors.* Expert users can already benefit from the previous class of tutors. Nevertheless, to achieve a significative production gain, they need more intelligent tutors implementing domain-specific theorem provers or able to perform complex computations. These tutors are not just plain implementations of tactics or decision procedures, but can be more complex software

agents interacting with third-parties software, such as proof-planners, CAS or theorem-provers.
To test the productivity impact of intelligent tutors, we have implemented a tutor that is interfaced with the HELM Search-Engine[12] and that is able to look for every theorem in the distributed library that can be applied to proceed in the proof. Even if the tutor deductive power is extremely limited[13], it is not unusual for the tutor to come up with precious hints that can save several minutes of work that would be spent in proving again already proven results or figuring out where the lemmas could have been stored in the library.

6 Conclusions and Future Work

In this paper we described a suggestion engine architecture for proof-assistants: the client (a proof-assistant) sends the current proof status to several distributed Web-Services (called tutors) that try to progress in the proof and, in case of success, send back an appropriate hint (a proof-plan) to the user. The user, that in the meantime was able to reason and progress in the proof, is notified with the hints and can decide to apply or ignore them. A broker is provided to decouple the clients and the tutors and to allow the client to locate and invoke the available remote services. The whole architecture is an instance of the MONET architecture for Mathematical Web-Services. It constitutes a reimplementation of the core features of the pioneering Ωmega-Ants system in the new Web-Services framework.

A running prototype has been implemented as part of the HELM project [5] and we already provide several tutors. Some of them are simple tutors that try to apply one or more tactics of the HELM Proof-Engine, which is also our client. We also have a much more complex tutor that is interfaced with the HELM Search-Engine and looks for lemmas that can be directly applied.

Future works comprise the implementation of new features and tutors, and the embedding of the system in larger test cases. For instance, one interesting case study would be interfacing a CAS as Maple to the H-Bugs broker, developing at the same time a tutor that implements the Field tactic of Coq, which proves the equality of two expressions in an abstract field by reducing both members to the same normal form. CASs can produce several compact normal forms, which are particularly informative to the user and that may suggest how to proceed in a proof. Unfortunately, CASs do not provide any certificate about the correctness of the simplification. On the contrary, the Field tactic certifies the equality of two expressions, but produces normal forms that are hardly a simplification of the original formula. The benefits for the CAS would be obtained by using the Field tutor to certify the CAS simplifications, proving that the Field normal form of an expression is preserved by the simplification. More advanced tutors

[12] `http://helm.cs.unibo.it/library.html`

[13] We do not attempt to check if the new goals obtained applying a lemma can be automatically proved or, even better, automatically disproved to reject the lemma.

could exploit the CAS to reduce the goal to compact normal forms [10], making the Field tutor certify the simplification according to the skeptical approach.

We have many plans for further developing both the H-Bugs architecture and our prototype. Interesting results could be obtained augmenting the informative content of each suggestion. We can for example modify the broker so that also negative results are sent back to the client. Those negative suggestions could be reflected in the user interface by deactivating commands to narrow the choice of tactics available to the user. This approach could be interesting especially for novice users, but requires the client to increase their level of trust in the other actors.

We plan also to add some rating mechanism to the architecture. A first improvement in this direction could be distinguishing between hints that, when applied, are able to completely close one or more goals, and tactics that progress in the proof by reducing one or more goals to new goals: since the new goals can be false, the user can be forced later on to backtrack.

Other heuristics and or measures could be added to rate hints and show them to the user in a particular order: an interesting one could be a measure that try to minimize the size of the generated proof, privileging therefore non-overkilling solutions [13].

We are also considering to follow the Ωmega-Ants path adding "recursion" to the system so that the proof status resulting from the application of old hints are cached somewhere and could be used as a starting point for new hint searches. The approach is interesting, but it represents a big shift towards automatic theorem proving: thus we must consider if it is worth the effort given the increasing availability of automation in proof assistants tactics and the ongoing development of Web-Services based on already existent and well developed theorem provers.

Even if not strictly part of the H-Bugs architecture, the graphical user interface (GUI) of our prototype needs a lot of improvement if we want it to be really usable by novices. In particular, a critical issue is avoiding continuous distractions for the user determined by the hints that are asynchronously pushed to her.

Our Web-Services still lack a real integration in the MONET architecture, since we do not provide the different ontologies to describe our problems, solutions, queries, and services. In the short term, completing this task could provide a significative feedback to the MONET consortium and would enlarge the current set of available MONET actors on the Web. In the long term, new more intelligent tutors could be developed on top of already existent MONET Web-Services.

To conclude, H-Bugs is a nice experiment meant to understand whether the current Web-Services technology is mature enough to have a concrete and useful impact on the daily work of proof-assistants users. So far, only the tutor that is interfaced with the HELM Search-Engine has effectively increased the productivity of experts users. The usefulness of the tutors developed for beginners, instead, need further assessment.

References

1. Web Services Glossary, W3C Working Draft, 14 May 2003.
   `http://www.w3.org/TR/2003/WD-ws-gloss-20030514/`
2. Web Services Description Language (WSDL) Version 1.2: Bindings, W3C Working Draft, 24 January 2003.
   `http://www.w3.org/TR/wsdl12-bindings/`
3. A. Armando, D. Zini. Interfacing Computer Algebra and Deduction Systems via the Logic Broker Architecture. In Proceedings of the Eighth Calculemus symphosium, St. Andrews, Scotland, 6–7 August 2000.
4. O. Caprotti. Symbolic Evaluator Service. Project Report of the MathBrocker Project, RISC-Linz, Johannes Kepler University, Linz, Austria, May 2002.
5. A. Asperti, F. Guidi, L. Padovani, C. Sacerdoti Coen, I. Schena. Mathematical Knowledge Management in HELM. In Annals of Mathematics and Artificial Intelligence, 38(1): 27–46, May 2003.
6. C. Benzmüller, V. Sorge. O-Ants – An Open Approach at Combining Interactive and Automated Theorem Proving. In M. Kerber and M. Kohlhase (eds.), Integration of Symbolic and Mechanized Reasoning, pp. 81–97, 2000.
7. C. Benzmüller, M. Jamnik, M. Kerber, V. Sorge. Agent-based Mathematical Reasoning. In A. Armando and T. Jebelean (eds.), Electronic Notes in Theoretical Computer Science, (1999) 23(3), Elsevier.
8. C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, K. Konrad, E. Melis, A. Meier, W. Schaarschmidt, J. Siekmann, V. Sorge. OMEGA: Towards a Mathematical Assistant. In W. McCune (ed), Proceedings of the 14th Conference on Automated Deduction (CADE-14), Springer LNAI vol. 1249, pp. 252–255, Townsville, Australia, 1997.
9. S. Boutin. Using reflection to build efficient and certified decision procedures. In Martin Abadi and Takahashi Ito, editors, TACS'97, volume 1281. LNCS, Springer-Verlag, 1997.
10. David Delahaye, Micaela Mayero. A Maple Mode for Coq. Contribution to the Coq library.
    `htpp://coq.inria.fr/contribs/MapleMode.html`
11. The MONET Consortium, MONET Architecture Overview, Public Deliverable D04 of the MONET Project.
    `http://monet.nag.co.uk/cocoon/monet/publicsdocs/monet-overview.pdf`
12. C. Sacerdoti Coen. Exportation Module, MoWGLI Deliverable D2.a.
    `http://mowgli.cs.unibo.it/html\_no\_frames/deliverables/transformation/d2a.html`
13. C. Sacerdoti Coen. Tactics in Modern Proof-Assistants: the Bad Habit of Overkilling. In Supplementary Proceedings of the 14th International Conference TPHOLS 2001, pp. 352–367, Edinburgh.
14. S. Zacchiroli. *Web services per il supporto alla dimostrazione interattiva*, Master Thesis, University of Bologna, 2002.
15. J. Zimmer and M. Kohlhase. System Description: The MathWeb Software Bus for Distributed Mathematical Reasoning. In Proceedings of the 18th International Conference on Automated Deduction CADE 18, LNAI 2392, Springer Verlag, 2002.
16. R. Zippel. The MathBus. In Workshop on Internet Accessible Mathematical Computation at ISSAC'99, Vancouver, Canada, July 28–31, 1999.

Trustable Communication Between Mathematics Systems*

Jacques Carette, William M. Farmer, and Jérémie Wajs**

McMaster University
Hamilton, Ontario, Canada

Abstract. This paper presents a rigorous, unified framework for facilitating communication between mathematics systems. A mathematics system is given one or more interfaces which offer deductive and computational services to other mathematics systems. To achieve communication between systems, a client interface is linked to a server interface by an asymmetric connection consisting of a pair of translations. Answers to requests are trustable in the sense that they are correct provided a small set of prescribed conditions are satisfied. The framework is robust with respect to interface extension and can process requests for abstract services, where the server interface is not fully specified.

Keywords: Mechanized mathematics, computer theorem proving, computer algebra, intersystem communication, knowledge representation.

1 Introduction

Current mechanized mathematics systems (MMSs), by and large, fall into one of three camps: numerics-based (like Matlab, Octave, Scilab, etc), symbolic (Maple, Mathematica, MuPAD, etc), and theorem provers (Coq, HOL, IMPS, Isabelle, Nqthm, Nuprl, Otter, PVS, etc). Each has its strong points, although many are more often bemoaned for their weaknesses. These weaknesses are all the more frustrating for users as one system's weakness is frequently another's strength. An increasing majority of users are becoming agnostic in their choice of MMSs, worrying more about getting a particular task done than whether one übersystem can do it all. Furthermore, it is important to remark that the expertise needed to build each kind of system is markedly different for all three flavors. Although there have been some efforts at making some of these MMSs broader, familiarity with them quickly dispels any notion that this dabbling is particularly successful. A wiser approach, at least in the medium term, is to construct a larger system out of trusted specialized pieces.

In simple terms, the problem we wish to address, illustrated in Figure 1, is the following: if system A needs access to a certain functionality f which it does not currently implement, but a service providing this functionality is offered by system B, then A should be able to send a request to B containing a translation of its exact problem into the language of B, wait for B to perform the service, and then finally receive an answer in its own language.

* This research was supported by Bell Canada and MITACS.

** {carette,wmfarmer,wajs}@mcmaster.ca

$$\begin{array}{ccc} A\text{-problem} & \xrightarrow{\text{translation}} & B\text{-problem} \\ {\scriptstyle f}\big\downarrow & & \big\downarrow{\scriptstyle B\text{-service}} \\ A\text{-answer} & \xleftarrow{\text{translation}} & B\text{-answer} \end{array}$$

Fig. 1. The basic communication problem

Informally, we wish to think of "perform f" as a request, the pair of translations above as a connection, and the set of available functions from B-problems to B-answers as B's services. We then want to assert that *meaningful communication* happens when the diagram above commutes.

In this paper we present a unified framework which clearly defines these various concepts (*interfaces*, *services*, *connections*, *requests*, and *answers*) in precise mathematical terms. The overarching concern is that of *trust*: when one system requests a service from another, can it trust the result it gets back? Certainly any system which purports to be trustable must also insist that any communication it makes to another system satisfies the same requirements. We have not generally addressed the concept of *usefulness* of the resulting communication, as we are not aware of any generally accepted mathematical definition of that concept.

Certainly examples of *useful* communication between systems abound! Commercial system builders are definitely convinced of this fact, as evidenced by Mathematica's J/Link, Maple's Matlab package, Matlab's Symbolic Toolbox, and so on.

For example, polynomial arithmetic is frequently a necessary step in a proof; typical theorem provers will, at best, implement this using rewrite rules, which are at least an order of magnitude slower than implementations by Computer Algebra Systems (CASs) [7]. In the opposite direction, closed-form integration of even simple expressions containing parameters involves complex algorithms but also complex side conditions which must be verified, forcing a CAS to call a theorem prover (see [1] and the references therein).

We consider old obstacles (issues of transport and syntax) to be essentially solved by common technologies (TCP/IP, sockets, XML, etc). What remains to be solved adequately is the problem of semantics. Referring back to Figure 1, it should be clear that describing each arrow, in all cases and for all possible services, is nontrivial. To achieve our aim of *trustability*, this issue is inescapable. To a lesser extent, there is also a problem of interpretability: even if the answer makes sense in system A, is it "the" answer? The notion of "the" answer in a theorem proving system is qualitatively different than in a system centered on numerical analysis, even though both are rigorously and uniquely defined.

More discussion can be found in the long version [9] of this paper.

The rest of the paper is organized as follows: In section 2, we look at previous related proposals. In section 3, we give definitions for the underlying theory necessary to the presentation of our framework. In section 4, we give a simple framework for communication between MMSs. In section 5, we discuss additional obstacles in achieving communication in real cases, and show how to refine the framework presented in section 4 to address some of those obstacles. In section 6, we talk about specification of requests and services. Finally, we conclude in section 7.

2 Previous Proposals

Several attempts at addressing the problem of communication between MMSs have been made. We can classify them into two categories: the first category consists of work that attempts to deal with the problem in general. The second category consists of *ad hoc* solutions. We review important members of each category below.

General Solutions The OpenMath project [10] claims to provide a common platform for communication between various mathematics systems. However, while it provides a common syntax, it fails in our view to specify a semantics for that syntax, which is a major drawback when trying to make mathematics systems based on different logics communicate. In other words, there are too many implicit assumptions behind OpenMath's version of semantics for it to apply outside the narrow (but useful) realm of standard operations between the standard CASs.

OMDoc [15] constitutes a refinement to the OpenMath approach: it recognizes the need for semantics, and introduces them through a notion of *theories*. However, OMDoc does not seem to address the actual mechanics of getting different systems to communicate as much as it provides a common language (syntax + semantics) for them to do so. Nevertheless, OMDoc could be extended to handle the concepts of our framework: interfaces, services, connections, requests, and answers.

The Ω-MKRP [14] approach argues that explicit proofs are needed and that "external" systems cannot be trusted. This seems very impractical.

The OMSCS (Open Mechanized Symbolic Computation Systems) [8] work provides an architecture used to formally specify automated theorem provers and CASs and to formally integrate them. However, it does not seem to address the issues of trust or extending theories.

Armando and Zini's Logic Broker Architecture [2], defines a general framework for communication between MMSs. This approach is conceptually very similar to ours. It defines interfaces for MMSs and uses a *Logic Broker* (LB) to achieve communication between systems. The LB includes facilities for translation of requests and meaning-preserving translation of answers (thus addressing the question of trust), as well as (in theory) a logical specification matcher to match requests to services offered. However, we believe that this architecture does not support extending theories well, which we will show can be achieved effectively by our approach.

The new European MOWGLI project [3], which aims at providing a common machine-understandable (semantics-based) representation of mathematical knowledge and a platform to exploit it, likely fits here too.

Ad-hoc approaches In many such cases in the literature, only unidirectional cooperation exists: one system acts as a master, generating requests, while the other one serves as a slave, fulfilling those requests. This includes Howe's work on embedding an HOL theory into Nuprl [13], Ballarin and Paulson's work on using the Sumit library for proofs in Isabelle [5, 7], and Ballarin, Homann, and Calmet's work on an interface between Isabelle and Maple [6]. Ballarin and Paulson's work clearly identifies the issue of trust, and distinguishes between trustable results, for which a formal proof exists, and ad hoc results, based on approximations.

Another more complex *ad hoc* case, intended for bidirectional cooperation, is Harrison and Théry's work on combining HOL and Maple [12]. Similarly to Ballarin and Paulson, they classify the systems by *degree of trust*, for example trusting results proved by HOL while checking results given by Maple.

All these *ad hoc* solutions have the major drawback of not seeking generality. Howe, for instance, does not attempt to make HOL and Nuprl *communicate* as much as he attempts to *embed* an HOL theory into Nuprl. Why should the machinery for HOL be duplicated in Nuprl when it already exists in HOL itself? In addition, this approach is not valid when the system to be integrated is a black box. Our approach enables one MMS to use another MMS's services without, first, having to reproduce them, and second, having to know in detail how they work. We will show how it addresses the issue of trust, and eliminates the need to verify every single result (which can be painfully burdensome).

3 Biform Theories

At the heart of this work lies the notion of a "biform theory", which is the basis for FFMM, a Formal Framework for Managing Mathematics [11]. Informally, a biform theory is simultaneously an axiomatic and an algorithmic theory. Most of the definitions given here are simplified versions of definitions given in [11].

A *language* is a set of typed expressions. The types include $*$, which denotes the type of truth values. A *formula* is an expression of type $*$. For a formula A of a language L, $\neg A$, the negation of A, is also a formula of L. A *logic* is a set of languages with a notion of logical consequence. If $\mathbf{K}$ is a logic, L is a language of $\mathbf{K}$, and $\Sigma \cup \{A\}$ is a set of formulas of L, then $\Sigma \models_{\mathbf{K}} A$ means that A is a logical consequence of Σ in $\mathbf{K}$.

Let L_i be a language for $i = 1, 2$. A *transformer* Π from L_1 to L_2 is an algorithm that implements a partial function $\pi : L_1 \rightharpoonup L_2$. For $E \in L_1$, let $\Pi(E)$ mean $\pi(E)$, and let $\mathsf{dom}(\Pi)$ denote the domain of π, i.e., the subset of L_1 on which π is defined.

A *formuloid* of a language L is a pair $\theta = (\Pi, M)$ where:

1. Π is a transformer from L to L.
2. M is a function that maps each $E \in \mathsf{dom}(\Pi)$ to a formula of L.

M is intended to give the *meaning* of applying Π to an expression E. $M(E)$ usually relates the input E to the output $\Pi(E)$ in some way; for many transformers, $M(E)$ is the equation $E = \Pi(E)$, which says that Π transforms E into an expression with the same value as E itself.

The *span* of θ, written $\mathsf{span}(\theta)$, is the set $\{M(E) \mid E \in \mathsf{dom}(\Pi)\}$ of formulas of L. Thus a formuloid has both an *axiomatic meaning*—its span—and an *algorithmic meaning*—its transformer. The purpose of its span is to assert the truth of a set of formulas, while its transformer is meant to be a deduction or computation rule.

A *biform theory* is a tuple $T = (\mathbf{K}, L, \Gamma)$ where:

1. $\mathbf{K}$ is a logic called the *logic* of T.
2. L is a language of $\mathbf{K}$ called the *language* of T.
3. Γ is a set of formuloids of L called the *axiomoids* of T.

The *span* of T, written $\mathsf{span}(T)$, is the union of the spans of the axiomoids of T, i.e., $\bigcup_{\theta \in \Gamma} \mathsf{span}(\theta)$. A is an *axiom* of T if $A \in \mathsf{span}(T)$. A is a *theorem* of T, written $T \models A$, if $\mathsf{span}(T) \models_{\mathbf{K}} A$. A *theoremoid* of T is a formuloid θ of L such that, for each $A \in \mathsf{span}(\theta)$, $T \models A$. Obviously, each axiomoid of T is also a theoremoid of T. An axiomoid is a generalization of an axiom; an individual axiom A (in the usual sense) can be represented by an axiomoid (Π, M) such that $\mathsf{dom}(\Pi) = \{A\}$ and $M(A) = A$.

T can be viewed as simultaneously both an *axiomatic theory* and an *algorithmic theory.* The axiomatic theory is represented by

$$T_{\mathrm{axm}} = (\mathbf{K}, L, \{M(E) \mid (\Pi, M) \in \Gamma \text{ for some } \Pi \text{ and } E \in \mathsf{dom}(\Pi)\}),$$

and the algorithmic theory is represented by

$$T_{\mathrm{alg}} = (\mathbf{K}, L, \{\Pi \mid (\Pi, M) \in \Gamma \text{ for some } M\}).$$

Let $T_i = (\mathbf{K}, L_i, \Gamma_i)$ be a biform theory for $i = 1, 2$. T_2 is an *extension* of T_1, written $T_1 \leq T_2$, if $L_1 \subseteq L_2$ and $\Gamma_1 \subseteq \Gamma_2$. T_2 is a *conservative extension* of T_1, written $T_1 \trianglelefteq T_2$, if $T_1 \leq T_2$ and, for all formulas A of L_1, if $T_2 \models A$, then $T_1 \models A$. Note that $\leq$ and $\trianglelefteq$ are partial orders.

Let $\mathbf{K}_i$ be a logic and $T_i = (\mathbf{K}_i, L_i, \Gamma_i)$ be a biform theory for $i = 1, 2$. A *translation* from T_1 to T_2 is a transformer Φ from L_1 to L_2 that:

1. Respects types, i.e., if E_1 and E_2 are expressions in L_1 of the same type and $\Phi(E_1)$ and $\Phi(E_2)$ are defined, then $\Phi(E_1)$ and $\Phi(E_2)$ are also of the same type.
2. Respects negation, i.e., if A is a formula in L_1 and $\Phi(A)$ is defined, then $\Phi(\neg A) = \neg\Phi(A)$.

T_1 and T_2 are called the *source theory* and the *target theory* of Φ, respectively. Φ is *total* if $\Phi(E)$ is defined for each $E \in L_1$. Φ *fixes* a language L if $\Phi(E) = E$ for each $E \in L$.

An *interpretation* of T_1 in T_2 is a total translation Φ from T_1 to T_2 such that, for all formulas $A \in L_1$, if $T_1 \models A$, then $T_2 \models \Phi(A)$. An interpretation thus maps theorems to theorems. (Since any translation respects negation, an interpretation also maps negated theorems to negated theorems.) A *retraction* from T_2 to T_1 is an interpretation Φ of T_2 in T_1 such that $T_1 \leq T_2$ and Φ fixes L_1.

Proposition 1. *If Φ is a retraction from T_2 to T_1, then $T_1 \trianglelefteq T_2$.*

Proof. Let A be a formula of the language of T_1 such that $T_2 \models A$. We must show that $T_1 \models A$. By definition, (1) Φ is an interpretation of T_2 in T_1 and (2) Φ fixes the language of T_1. (1) implies that $T_1 \models \Phi(A)$, and (2) implies $\Phi(A) = A$. Therefore, $T_1 \models A$. □

4 A Simple Communication Framework

We now present a simple communication framework, based on the theoretical notions presented in the previous section, that addresses the problem presented in Figure 1. The framework formalizes the notions we mentioned in the introduction: interface, service, connection, request, and answer. As we will show after this section, the framework does not address some important obstacles to effective communication between MMSs. A refined framework, which is more practical and which generalizes this simple framework, is presented in section 5.

An *interface* is a pair $I = (T, \mathcal{S})$ where:

1. T is a biform theory called the *theory* of I.
2. $\mathcal{S}$ is a set of theoremoids of T called the *services* of I.

As a theoremoid of T, a service of I is a formuloid whose span is a set of theorems of T and whose transformer is a sound deduction or computation rule for T.

Let $I_i = (T_i, \mathcal{S}_i)$ be an interface for $i = 1, 2$. A *connection* from I_1 to I_2 is a pair $C = (\mathsf{export}, \mathsf{import})$ where export is a translation from T_1 to T_2, and import is an interpretation of T_2 in T_1. I_1 and I_2 are respectively called the *client interface* and the *server interface* of C. export is for transporting problems from T_1 to T_2; it need not be meaning preserving. import transports solutions from T_2 to T_1; it must be meaning preserving.

An *informed request* is a tuple $R = (C, E, \theta)$ where:

1. $C = (\mathsf{export}, \mathsf{import})$ is a connection from $I_1 = (T_1, \mathcal{S}_1)$ to $I_2 = (T_2, \mathcal{S}_2)$.
2. E is an expression of the language of T_1.
3. $\theta = (\Pi, M) \in \mathcal{S}_2$.

The reason to call such a request *informed* is that it explicitly depends not only on the interface I_2 but on the theoremoid θ as well: we assume that I_1 "knows" about θ. We will come back to this point in section 5.

If $A = (\mathsf{import} \circ M \circ \mathsf{export})(E)$ is defined, it is the *answer* to R; otherwise the answer to R is undefined. When A is defined, it is a theorem:

Proposition 2. *Let R and A be as above. If A is defined, then $T_1 \models A$.*

Proof. Assume A is defined. Since θ is a theoremoid of T_2, $T_2 \models (M \circ \mathsf{export})(E)$, and then since import is an interpretation of T_2 in T_1, $T_1 \models (\mathsf{import} \circ M \circ \mathsf{export})(E)$. □

$$
\begin{tikzcd}
E \arrow[r, "\mathsf{export}"] \arrow[d, "?"'] & E' \arrow[d, "\theta"] \\
\mathsf{answer} & M(E') \arrow[l, "\mathsf{import}"]
\end{tikzcd}
$$

Fig. 2. Communication between two MMSs

Note that, if C and θ are not chosen well, A may be a useless theorem such as true or $E = E$.

The basic problem (Figure 1) is now addressed as shown in Figure 2. All that is necessary to perform this type of communication are interfaces for both systems and a connection between the two interfaces.

This takes care of the question of *trust* (should A believe the answer it receives from B?), so crucial to the general problem at hand. Whether an answer is correct depends on whether a translation is an interpretation and a service is a theoremoid. Thus an answer is trustworthy if the mechanisms for verifying interpretations and theoremoids are trustworthy.

Note also at this point that a given system may have many interfaces, each containing only one or a few services of that system. This approach allows us to consider trustable subsystems within a system and to use those subsystems in trustable communication. For example, while a result given by Maple cannot be fully trusted in general, many subparts of Maple are well encapsulated and could be proved correct.

Example using Decision Procedures

Suppose S_{hol} is a higher-order interactive theorem proving system with several implemented theories including COF, a theory of a complete ordered field. COF has one model up to isomorphism, namely, the real numbers with the usual operations such as $+$, $*$, and $<$. An exceedingly rich theory, COF is adequate for developing real analysis. Suppose also that S_{fol} is a first-order automated theorem proving system with several implemented theories equipped with decision procedures including PA, a theory of first-order Peano arithmetic. The theoremoids of PA include θ_+, a decision procedure for additive number theory (Presburger arithmetic), and θ_*, a decision procedure for multiplicative number theory (sometimes called Skolem arithmetic). The framework outlined above can be used to give S_{hol} access to the decision procedures in S_{fol}.

Let $I_1 = (\mathsf{COF}, \mathcal{S}_1)$ be an interface of S_{hol} and $I_2 = (\mathsf{PA}, \mathcal{S}_2)$ with $\{\theta_+, \theta_*\} \subseteq \mathcal{S}_2$ be an interface of S_{fol}. Also let $C = (\mathsf{export}, \mathsf{import})$ be the connection from I_1 to I_2 where export translates "first-order natural number formulas" of COF to formulas of PA and import is a standard interpretation of PA in COF. (Because COF satisfies Peano's (second-order) axioms for natural number arithmetic, (1) export is not an interpretation and (2) import exists.) C offers a way of deciding in COF many statements about the natural numbers using the two decision procedures θ_+ and θ_*, both of which are nontrivial to implement. See [9] for further details.

5 A Refined Communication Framework

There are several obstacles to effectively employing the simple framework presented in the previous section. In this section, three obstacles involving connections are addressed.

The first obstacle is that constructing connections between interfaces is a challenging task, especially when the biform theories of the interfaces are based on different logics. The export translation of a connection must satisfy a syntactic condition, but the import interpretation must satisfy both a syntactic and semantic condition. As a general principle, it is easier to construct a translation or interpretation Φ if the "primitive basis" of its source theory T_1 (the primitive symbols and axiomoids of T_1) is small.

The second obstacle is that translating an expression E using the export translation or the import interpretation of a connection may result in an expression much larger than E. As a general principle, it is easier to construct a translation or interpretation Φ without this kind of size explosion if its target theory T_2 contains a rich set of defined symbols.

The third obstacle is that the theory S of an MMS behind the biform theory T of an interface is likely to be enriched with defined symbols over time. Defining a symbol in S will have the effect of extending T to a new theory T'. However, an interpretation Φ of T will not be an interpretation of T' because Φ will not be defined on expressions of T' containing the new defined symbol. As a result, any connection to an interface of the form $(T, \mathcal{S})$ will be broken by the definition of the new symbol.

These three obstacles can be addressed by using a "conservative stack" in place of a biform theory in the definition of an interface. Interface, connection, informed request, and answer are redefined. The resulting refined framework is a generalized version of the simple framework.

A *conservative stack* is a pair $\Sigma = (\tau, \rho)$ of sequences where:

1. $\tau = \langle T_0, \ldots, T_n \rangle$ is a finite sequence of biform theories such that, for all i with $0 \leq i < n$, $T_i \leq T_{i+1}$. T_n is called the *theory* of Σ.
2. $\rho = \langle \Phi_1, \ldots, \Phi_n \rangle$ is a finite sequence of translations such that, for all i with $0 < i \leq n$, Φ_i is a retraction from T_i to T_{i-1}.

Notice that, by Proposition 1, the sequence ρ of retractions implies that τ is a "stack" of conservative extensions, i.e., $T_0 \trianglelefteq \cdots \trianglelefteq T_n$.

An *interface* is a pair $I = (\Sigma, \mathcal{S})$ where Σ is a conservative stack and $\mathcal{S}$ is a set of theoremoids of the theory of Σ called the *services* of I.

Let $I_i = ((\tau_i, \rho_i), \mathcal{S}_i)$ be an interface with $\tau_i = \langle T^i_0, \ldots, T^i_{n_i} \rangle$ for $i = 1, 2$. A *connection* C from I_1 to I_2 is a pair $(\mathsf{export}, \mathsf{import})$ where:

1. export is a translation from U^1 to V^2.
2. import is an interpretation of U^2 in V^1.
3. U^1 and V^1 are members of τ_1.
4. U^2 and V^2 are members of τ_2

Let Φ_i be the composition of elements of ρ_i from $T^i_{n_i}$ to U^i for $i = 1, 2$. It is easy to see that Φ_i is a retraction from $T^i_{n_i}$ to U^i for $i = 1, 2$. $(\mathsf{export} \circ \Phi_1, \mathsf{import} \circ \Phi_2)$ is a connection from $(T^1_{n_1}, \mathcal{S}_1)$ to $(T^2_{n_2}, \mathcal{S}_2)$ in the simple framework even if $U^1 \neq V^1$ or $U^2 \neq V^2$.

An *informed request* is a tuple $R = (C, E, \theta)$ where:

1. C is a connection from I_1 to I_2 as defined above.
2. E is an expression of the language of $T^1_{n_1}$, the theory of I_1.
3. $\theta = (\Pi, M) \in \mathcal{S}_2$.

If $A = (\mathsf{import} \circ \Phi_2 \circ M \circ \mathsf{export} \circ \Phi_1)(E)$ is defined (where Φ_1 and Φ_2 are defined as above), it is the *answer* to R; otherwise the answer to R is undefined. When A is defined, it is a theorem:

Proposition 3. *Let R and A be as above. If A is defined, then $V^1 \models A$.*

Proof. Assume that A is defined. Since θ is a theoremoid of $T^2_{n_2}$, the theory of I_2, $T^2_{n_2} \models (M \circ \mathsf{export} \circ \Phi_1)(E)$, and since Φ_2 is a retraction from $T^2_{n_2}$ to U^2, $U^2 \models (\Phi_2 \circ M \circ \mathsf{export} \circ \Phi_1)(E)$. Since import is an interpretation of U^2 in V^1, we conclude that $V^1 \models A$. $\square$

The refined framework facilitates the construction of a translation or interpretation Φ between two interfaces I_1 and I_2 by allowing the source theory of Φ to be chosen from the lower part of the conservative stack of I_1 and the target theory of Φ to be chosen from the upper part of the conservative stack of I_2 (addressing the first and second obstacles discussed above). If a conservative stack Σ is extended to a larger conservative stack Σ', then Σ can be freely replaced with Σ' without compromising any existing interfaces or connections (addressing the third obstacle).

6 Specifying Requests and Services

Until now, we assumed that system A “magically” knows that it wants to use service θ of system B. However, in a more general setting, one would want to specify a request (like *evaluate this computation*), and pass that specification on to some entity able to match it to an available service.

Thus, instead of dealing with services of I_2, we need to deal with some specification S corresponding to some function $f : L_1 \to L_1$ (a computational transformer) associated with a “virtual service” θ_1. Given S, the task then becomes one of *finding* an informed request such that our communication diagram commutes. In theory, this is what we understand that Armando and Zini's LS Matcher [2] is somehow supposed to perform, although its task is never defined precisely.

Let us define *reachable services* as those computational theoremoids θ_2 of T_2 that can be given a complete specification in some meta-language Spec. We could, for example, use CASL [4], *Z* [17] or *Specware* [16] for this task. In other words, we wish to define services (and requests) implicitly, allowing nonconstructive definitions as well. Note that we specifically exclude theoremoids that cannot be finitely axiomatized in Spec. Symmetrically to reachable services, we define (brokered) *requests* as those virtual services θ_1 of I_1 which can be specified completely in Spec.

We then need to solve the specification matching problem: given a pair (S_1, S_2) of specifications for θ_1 and θ_2, does there exist a connection C such that our communication diagram commutes?

Even in the simplest possible case where both systems are the same, this problem can still be quite difficult unless great pains are taken to specify each system's services in a very uniform manner. However the situation is far from hopeless: even though there are many different ways to specify that, for example, a particular function is a primality verification function (or an implementation thereof), the task of deciding that two such specifications are equivalent is considerably simpler than actually providing a provably correct implementation!

7 Conclusion

In this paper we have presented a mathematically rigorous framework for communicating mathematics between MMSs. This framework gives precise meanings to notions such as *(biform) theories*, *interfaces*, *services*, *connections*, *requests*, and *answers*. It addresses the issue of trust, which has been identified as a central issue in intersystem communication in related papers, by using *interpretations* (meaning-preserving translations) to communicate answers. It also provides facilities for conservatively extending theories, allowing them to evolve as needed without needing to rebuild whole new interfaces or to drastically update connections.

We have defined precisely the problem of specification of services, and of logical specification matching. We are aware that any useful implementation of the ideas detailed in this paper would need to include such a facility, and we are working in that direction.

References

1. A. Adams, M. Dunstan, H. Gottliebsen, T. Kelsey, U. Martin, and S. Owre. Computer algebra meets automated theorem proving: Integrating Maple and PVS. In R. J. Boulton and P. B. Jackson, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2001)*, volume 2152 of *LNCS*, pages 27–42. Springer-Verlag, 2001.
2. A. Armando and D. Zini. Interfacing computer algebra and deduction systems via the logic broker architecture. In M. Kerber and M. Kohlhase, editors, *Symbolic Computation and Automated Reasoning (CALCULEMUS-2000)*, pages 49–64. A. K. Peters, 2001.
3. A. Asperti and B. Wegner. MOWGLI — a new approach for the content description in digital documents. In *Ninth International Conference on Electronic Resources and the Social Role of Libraries in the Future*, Autonomous Republic of Crimea, Ukraine, 2002.
4. E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki. CASL: The Common Algebraic Specification Language. *Theoretical Computer Science*, 286:153–196, 2002.

5. C. Ballarin. *Computer Algebra and Theorem Proving.* PhD thesis, Cambridge University, 1999.
6. C. Ballarin, K. Homann, and J. Calmet. Theorems and algorithms: An interface between Isabelle and Maple. In *International Symposium on Symbolic & Algebraic Computation (ISSAC-95)*, pages 150–157, 1995.
7. C. Ballarin and L. C. Paulson. A pragmatic approach to extending provers by computer algebra - with applications to coding theory. *Fundamenta Informaticae*, 39:1–20, 1999.
8. P. G. Bertoli, J. Calmet, F. Giunchiglia, and K. Homann. Specification and integration of theorem provers and computer algebra systems. *Fundamenta Informaticae*, 39:39–57, 1999.
9. J. Carette, W. M. Farmer, and J. Wajs. Trustable communication between mathematics systems. Technical report, McMaster University, 2003.
10. S. Dalmas, M. Gaëtano, and S. M. Watt. An OpenMath 1.0 implementation. In *International Symposium on Symbolic & Algrebraic Computation (ISSAC-97)*, pages 241–248, 1997.
11. W. M. Farmer and M. v. Mohrenschildt. An overview of a Formal Framework for Managing Mathematics. *Annals of Mathematics and Artificial Intelligence*, 38:165–191, 2003.
12. J. Harrison and L. Théry. A skeptic's approach to combining HOL and Maple. *Journal of Automated Reasoning*, 21:279–294, 1998.
13. D. J. Howe. Importing mathematics from HOL into Nuprl. In J. Von Wright et al., editors, *Theorem Proving in Higher Order Logics (TPHOLs 1996)*, volume 1125 of *LNCS*, pages 267–282. Springer-Verlag, 1996.
14. M. Kerber, M. Kohlhase, and V. Sorge. An integration of mechanised reasoning and computer algebra that respects explicit proofs. Technical Report CSRP-96-9, University of Birmingham, 1996.
15. M. Kohlhase. OMDoc: An open markup format for mathematical documents (version 1.1). Technical report, Carnegie Mellon University, 2002.
16. Y. V. Srinivas and R. Jullig. Specware: Formal support for composing software. In *Mathematics of Program Construction*, pages 399–422, 1995.
17. J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof.* Series in Computer Science. Prentice Hall, 1996.

System Description: Analytica 2

Edmund Clarke, Michael Kohlhase, Joël Ouaknine, Klaus Sutner

Carnegie Mellon University
{emc|kohlhase|ouaknine|sutner}@cs.cmu.edu

Abstract. The ANALYTICA system is a theorem proving system for 19^{th} century mathematics written on top of the *Mathematica* computer algebra system. It was developed in the early 1990's by X. Zhao and E. Clarke and has since been dormant. We describe recent work to resurrect the theorem prover and port it to newer versions of *Mathematica*. The new system ANALYTICA 2 can still prove the same theorems, but has been significantly cleaned up. The code has been restructured and documented, the declarative knowledge has been separated from a logical kernel, and the system is being made available as a MATHWEB service.

1 Introduction

The ANALYTICA system [CZ92,BCZ98] is a theorem proving system for 19^{th} century mathematics.

It has been able to prove theorems from elementary calculus and number theory, including a proof of the Bernstein approximation theorem and the theorems and examples in the second chapter in Ramanujan's Collected Work [Ber85,CZ92]. The system was developed in the early 1990's by Xudong Zhao and Edmund Clarke and has since been dormant.

ANALYTICA is written on top of the *Mathematica* computer algebra system [Wol02], a large commercial computer algebra system that offers a highly developed document-centered front-end that facilitates communication with the kernel and that allows for the development of multi-modal electronic documents, so-called notebooks, that can contain code, text, graphics, and data. Notebooks can render mathematical formulae in near-typeset quality. Moreover, Notebooks are symbolic structures that can be manipulated by the *Mathematica* kernel like any other symbolic expression in the system. They can also be exported in LaTeX and MATHML format. We suggest that this computational environment naturally supports the design and implementation of fairly complicated software systems using symbolic computation. A description of a similar effort in the area of computational automata theory can be found in [Sut02].

2 Porting the Code Base to *Mathematica* Version 5

ANALYTICA was originally written for *Mathematica* version 1.2, which lacked many of the features of current versions of the product. In particular, no graphical front-end was available and all communication to the kernel was handled by a text-based interface similar to a command shell. In our work on the ANALYTICA prover we make substantial use of four new capabilities of *Mathematica*: the

notebook front-end We have used the front-end in the documentation of the code base, and as user interface: As formula output in the *Mathematica* frontend approaches that of TEX and notebooks supports a powerful folding operation, ANALYTICA's original LATEX output routines for proofs are now obsolete and were deleted from ANALYTICA.

external system interface JLINK is used for interfacing to knowledge exchange formats like OMDOC (see Section 3).

native XML processing capabilities are used heavily in communication with XML based services.

added symbolic computation capabilities in the *Mathematica* kernel: For instance, support for symbolic summation and trigonometric simplification has dramatically improved in *Mathematica* since version 1.2. Nonetheless, we have retained existing ANALYTICA modules for these areas as plug-ins, loadable on demand. These implementations are transparent to the theorem prover and can thus be used to document proofs and computations that would be opaque if carried out by *Mathematica*'s built-in version.

The first step was to convert the formerly 50 plus source files into two large notebooks, one each for the prover and knowledge base parts (see Section 3). Code for the prover is represented using a special SOURCE style sheet that tightly integrates the actual *Mathematica* code with accompanying documentation, examples and test code. From the SOURCE style notebook one can automatically generate files that augment the *Mathematica* help browser and provide online help for the ANALYTICA system. From the same source document one can also extract pure code files that can be bundled together with the online documentation into an add-on package (see [CKOS03] for details). Installation of this package is very straightforward, and requires no more than to copy the package files to the appropriate place in the *Mathematica* file structure.

3 Separating Mathematical Knowledge from Code

There are two kinds of code in ANALYTICA: the program code and mathematical knowledge used in proof search. To separate causes and make ANALYTICA easier to port to other mathematical domains, these are separated in ANALYTICA2. Originally, the mathematical knowledge used in ANALYTICA was represented as the following *Mathematica* code.

```
(* Rules for simplifying expressions involving the  absolute value function. *)
UnProtect[Abs];
Abs[a_ b_] := Abs[a] Abs[b];
Abs[a_^n_] := Abs[a]^n;
Protect[Abs];

(*  Local rule  used in simplification. *)
AbsRule = {Abs[a_] :> If[TrueQ[WeakSimplify[a >= 0]], a,
               If[TrueQ[WeakSimplify[a <= 0]], -a, Abs[Factor1[a]]]]};
```

The first block specifies some rewriting rules for the *Mathematica* symbol `Abs` that are subsequently used by *Mathematica*'s built-in simplifier. The second code block specifies a rewrite rule used in a special simplification engine in ANALYTICA. The correctness of the ANALYTICA system depends on a couple of hundreds of such rules.

These rules are now collected in a notebook using as special Knowledge Representation style that captures the information implicit in the original code fragments. We have used a variant of the `nb2omdoc` transformer [Sut03] to transform KNOWLEDGE style notebooks into the OMDOC format (Open Mathematical DOCuments [Koh03]), an XML-based format for representing mathematical knowledge in the large. OMDOC can be used as a basis for communicating with other mathematical software systems and in particular, the MBASE mathematical knowledge base [KF01], which acts as an external knowledge repository for ANALYTICA 2 (see section 4).

In the transformation we have made explicit and thus documented the mathematical knowledge used in ANALYTICA. In the case of our example above, this is given by the 4 theorems:

#	formalization	the absolute value function ...
1	$\forall a, b.\lvert a \cdot b\rvert = \lvert a\rvert \cdot \lvert b\rvert$	commutes with multiplication
2	$\forall a, n.\lvert a^n\rvert = \lvert a\rvert^n$	commutes with exponentiation
3	$\forall a.a \geq 0 \Rightarrow \lvert a\rvert = a$	is the identity on $\mathbb{R}^+$
4	$\forall a.a \leq 0 \Rightarrow \lvert a\rvert = -a$	is the negative identity on $\mathbb{R}^-$

In the generated OMDOC representation, these theorems are represented in a special `assertion` element that combines the formalization in OPENMATH [CCAMC02] representation with the natural vernacular. Note that the *Mathematica* code fragments contain other information than the logical theorems, mostly of heuristic or computational nature, like the direction of the equation in the simplification rules. Therefore, the OMDOC representation also embeds the original *Mathematica* code. As *Mathematica* has a native XML (and thus OMDOC) parser, ANALYTICA can directly read OMDOC documents.

The main problem in the transformation to OMDOC is that the ANALYTICA logic is based on and uses many of the unique term representation features of the *Mathematica* language, which are geared for programming with mathematical objects, but whose logical foundations are insufficiently explored ([Mar03,Kut03] are recent exceptions).

For instance, functions in *Mathematica* are polyadic (they can have variable arities). To make this palatable to the user and programmer, *Mathematica* employs sequence variables in pattern matching. Consider for instance the following fragment from the definition for continuous functions.

```
Continuous[f_[a__], x_, x0_] :=
    Apply[ and, Map[ Function[z, Continuous[z, x, x0]], List[a]]] /; ContFunction[f];
```

The function `Continuous` takes three arguments, an expression e, a (bound) variable x, and a point x_0; it is true, if e is continuous at x_0 when viewed as a function in x. The interesting part is that the expression e is of the form $f(a_1, \ldots, a_n)$, where the variable `a` is a sequence variable that stands for the sequence $a_1, \ldots, a_n$[1].

Our OMDoc transformation currently treats sequence variables like arbitrary variables, and represents this as

$$\forall f, a, x, x_0 . \mathbb{C}(f(a), x, x_0) \Leftrightarrow \mathbb{C}^0(f) \wedge apply(\wedge, map(\lambda z \mathbb{C}(z, x, x_0))), list(a)$$

where we use $\mathbb{C}$ for `Continuous` and $\mathbb{C}^0$ for `ContFunction`. Of course, this is not a standard logical representation, and to communicate with other mathematical software systems we will need to translate this into more standard representations. One approach we are experimenting with at the moment is to encode sequence variables into higher-order logic with Currying, e.g. for communication with the Tps theorem prover for higher-order logic [ABI+96].

4 A MathWeb Interface for Analytica

For the communication with the MBase system, we have equipped Analytica with an XmlRpc interface, see [Com]. This allows Analytica to store the OMDoc-encoded knowledge externally and request the fragments needed for the proofs of the respective theorems. The XmlRpc interface is built on *Mathematica*'s JLink facility. and makes the Xml representations of the protocol documents available to *Mathematica*, whose native Xml facilities are used to convert them into Analytica's internal representations.

MBase is part of the MathWeb [ZK02] service infrastructure, which connects a wide-range of reasoning systems (*mathematical services*), such as automated theorem provers, (semi-)automated proof assistants, computer algebra systems, model generators, constraint solvers, human interaction units, and automated concept formation systems, by a common *mathematical software bus*. Reasoning systems integrated in the MathWeb can therefore new services to the pool of services, and can in turn use all services offered by other systems.

The next step in this development will to offer Analytica as a MathWeb service, making it possible to send problems in OMDoc form and receive OMDoc-encoded proofs in return. The main problem here lies in the *Mathematica*/Analytica logic as we have seen above. We plan to augment the proof output of Analytica to point to the justifying theorems to make Analytica proofs independent of the Analytica prover itself: Analytica does not currently produce proof objects; rather, a trace of the proof search is output as a side-effect. Eventually, we plan to supply proofs from first principles for all the knowledge used in the prover, so that Analytica proofs are grounded in axiomatics, as they should be for a theorem prover for mathematics.

[1] The postfix __ after the variable name marks `a` as a sequence variable for *Mathematica*. A single underscore marks a normal variable, and a triple one a possibly empty sequence variable

5 Conclusion

We have described a recent effort to port the code base of the ANALYTICA theorem prover to the newest version of the *Mathematica* language and to restructure it, so that can be extended to new mathematical areas. The knowledge part of ANALYTICA is translated to the OMDOC framework for mathematical knowledge representation. This general setup seems ideal for a knowledge-rich deduction component like the ANALYTICA theorem prover, and for the combination of computer algebra methods with proof engines.

References

[ABI+96] P. Andrews, M. Bishop, S. Issar, D. Nesmith, F. Pfenning, and Hongwei Xi. TPS: A theorem-proving system for classical type theory. *J. of Automated Reasoning*, 16:321–353, 1996.

[BCZ98] A. Bauer, E. Clarke, and X. Zhao. Analytica — an Experiment in Combining Theorem Proving and Symbolic Computation. *J. of Automated Reasoning*, 21(3):295–325, 1998.

[Ber85] B. C. Berndt. *Ramanujan's Notebooks, Part I*, pp. 25–43. Springer, 1985.

[CCAMC02] O. Caprotti, D. P. Carlisle, and eds. A. M. Cohen. The Open Math standard, version 1.1b. Technical report, The Open Math Society, `http://www.nag.co.uk/projects/OpenMath/omstd/`, 2002.

[CKOS03] E. Clarke, M. Kohlhase, J. Ouaknine, and K. Sutner. Resurrecting the ANALYTICA theorem prover. In *First QPQ Workshop on Deductive Software Components*, CADE-19, Miami, USA, 2003.

[Com] XML Remote Procedure Call Specification. `http://www.xmlrpc.com/`.

[CZ92] E. Clarke and X. Zhao. Combining symbolic computation and theorem proving: Some problems of Ramanujan. In D. Kapur, editor, *CADE-11*, volume 607 of *LNCS*, pages 66–78, 1992. Springer Verlag.

[KF01] M. Kohlhase and Andreas Franke. MBase: Representing knowledge and context for the integration of mathematical software systems. *Journal of Symbolic Computation*, 32(4):365–402, 2001.

[Koh03] M. Kohlhase. OMDOC an open markup format for mathematical documents (version 1.2). Technical report, Computer Science, Carnegie Mellon U., 2003. forthcoming.

[Kut03] T. Kutsia. Matching in flat theories. In UNIF2003 [UNI03].

[Mar03] M. Marin. Functional programming with sequence variables. In UNIF2003 [UNI03].

[Sut02] K. Sutner. `automata`, a hybrid system for computational automata theory. In J.-M. Champarnaud and D. Maurel, editors, *CIAA 2002*, pages 217–222, Tours, France, 2002.

[Sut03] Klaus Sutner. Converting mathematica notebooks to OMDOC. to appear in [Koh03], 2003.

[UNI03] 17^{th} *Workshop on Unification*, Valencia, Spain, 2003.

[Wol02] S. Wolfram. *The Mathematica Book*. Cambridge Univ. Press, 2002.

[ZK02] J. Zimmer and M. Kohlhase. System Description: The Mathweb Software Bus for Distributed Mathematical Reasoning. In A. Voronkov, editor, *CADE-18*, number 2392 in LNAI, pages 139–143. Springer, 2002.

A New Interface to PVS

A. A. Adams*

School of Systems Engineering, The University of Reading.
A.A.Adams@Rdg.ac.uk

Abstract. As part of an effort to produce a PVS server for MathWeb-SB, a basic MathWeb-SB was produced. From this prototype, we derived the specification of an interface to PVS for use as a proof server. The aim of this project is to develop first a server and then a client interface from PVS to MathWeb-SB. This intermediate stage of simply defining a new interface for PVS, it is hoped, will be of interest generally to the Calculemus community. Thus, this paper presents this new interface in both abstract and technical terms, and only in the "further work" section will the larger project of intergration into MathWeb-SB be considered.

1 Introduction

PVS (Prototype Verification System) [SOR] is a higher order theorem proving system developed at SRI International in Menlto Park, CA. Of particular interest to the Calculemus community is the development of a real analysis library by Gottliebsen [Got00]. This development offers the possibility of various "proof services" being offered by a PVS process requiring proof attempts of conjectures involving transcendental functions. These proof attempts can be anything from special-purpose PVS strategies (tactics), the gneral (quite powerful) PVS automatic proof strategies such as "grind", or even fully interactive proof attempts (for an experienced PVS user).

With this goal in mind, a prototype of a PVS/MathWeb-SB interface was produced in 2001. Using the lessons learned from this prototype, a new interface for using PVS in this manner has been developed. This paper describes this interface in abstract terms (first the full desired interface and then the current implemented parts of this) and then shows the initial implementation as Allegro Common Lisp code which can be loaded into an existing PVS installation. Note: once the interface has been tested it will be merged into the released code base of PVS.

2 Interface Description

PVS was designed and built to be used via an Emacs interface. Following initial experiments it was shown that it was possible to run the kernel of the system

* This work is supported by EU Grant Calculemus HPRN-CT-2000-00102 and UK EPSRC Grant GR/S15044.

directly, although a number of difficulties with the interface were identified. Not the least of these is that the input/output routines for the top-level of the ACL session were designed to take input from and return messages to, various Emacs buffers. As such, the resulting prototype interface was rather clumsy and quite sensitive to changes in the Emacs interface. Proof attempts were starteed using a function originally designed for a specific external program to use the PVS system as a back end, but this function was optimised for that very specific purpose and was difficult or impossible to control the way we desired. In addition, the prototype had to parse the printing of the PVS kernel during proof to check for a "Q.E.D." indicating a successful proof attempt.

So, we needed to define a new interface in PVS which was a single new function with various optional arguments to allow for a variety of behaviours. We are only interested in a functional specification for the interface.

- The default setting of the interface function will be to accept two arguments: a name and a conjecture. The conjecture will be parsed with respect to the "prelude" (the PVS default theory environment) and then the standard automatic proof strategy (grind) will be applied in an attempt to prove the conjecture.
- In some circumstances it is appropriate to attempt to prove the negaation of a submitted conjecture, should the attempt to prove it true fail. Note: PVS is a classical system, so double negation is not a problem. Thus, when a negative conjecture (NOT F) is submitted, the negation of this can be left as (NOT (NOT F)) rather than translated to F.
- There are four possible return values: *Proved*; *Disproved*; *Unknown* and *Unproved*. The difference between the last two is whether proof of a negation of the conjecture was attempted (Unproved) or not (Unknown).
- A variety of options to the interface are also useful:
  - An optional list of strategies for both positive and negative proof attempts.
  - A (list of) library name(s) (to add one of the distributed libraries to the prelude).
  - A flag to allow interactive proof rather than automated use of a strategy.

3 Interface Specification

The interface is implemented as a lisp function *prove-as-black-box*, with the following operation.

3.1 Arguments

Argument	Type	Default	
name	string		No whitespace allowed
lemma	string		PVS specification language
	Optional Arguments		
pos-strat	string or list of strings	nil	No whitespace allowed
neg-strat	string or list of strings	nil	No whitespace allowed
	Keyword Arguments		
library	string	nil	
file	string	nil	
interactive	t or nil	nil	

3.2 Error Reports

Error Type	Error Delimiters	Information
Parse Error	:(end-)pvs-parse-err	error location within lemma and PVS message
Typecheck Error	:(end-)pve-tc-err	error location within lemma and PVS message
Lisp Error	:(end-)pvs-lisp-err	Debug information and error report instructions

3.3 Return Values

Return Value	Interpretation
:proved	Positive Proof Attempt Successful
:disproved	Negative Proof Attempt Successful
:unknown	Unsuccessful, negation not attempted
:unproved	Unsuccessful, negation attempted

3.4 Side Effects

The global variable **pvs-black-box-proof** is set (with the usual PVS proof script syntax) to the proof resulting from the proof attempt. Thus, if there is a successful positive or negative proof attempt then this is stored here. Otherwise the final failed positive proof strategy is stored here.

3.5 Other Details

– Negated conjectures have "_negated" appended to the provided name.
– PVS generates Type Checking Conditions (TCCs) during typechecking. These are added in as conditionals to both positive and negative versions of the formula. Thus if TCC(F) is the TCCs generated by formula F then the positive conjecture is TTC(F)⇒F and the negated conjecture is TCC(F)⇒NOT(F).

– If the positive strategy list is empty and interactive is nil then the default strategy of *grind* is applied.
– If the negative strategy list is nil, no negation proof is attempted. If this is t rather than a list of strategies then the same strategy or strategies as used for the positive proof attempt is/are used.

4 Further Work

An implementation of the code as described above has been produced. It will be tested this summer (hopefully results will be available by the Calculemus meeting in Rome), and a new version of the MathWeb-SB half of the interface will be produced during this time. It is hoped that a MathWeb-SB interface for an automatic invocation of the PVS prover on a variety of libraries (in particular the real analysis library and continuity-checker of Gottliebsen) will be available late in 2003. Further work on making interactive use of the PVS prover via MathWeb-SB is expected in the future. In addition, we hope to develop a system allowing PVS to use other MathWeb-SB-enaled systems as oracles or black box provers/calculators in future.

5 Acknowledgements

The work on the PVS/MathWeb-SB interface is dependent on the cooperation of a number of other researchers:

– Sam Owre of SRI International.
– Jürgen Zimmer of The University of Edinburgh.
– Andreas Franke of Universität des Saarlandes.

Thanks and credit are due to them for their collaborative effort on this project.

References

[Got00] H. Gottliebsen. Transcendental Functions and Continuity Checking in PVS. pages 198–215. Springer-Verlag LNAI 1869, 2000.
[SOR] N. Shankar, S. Owre, and J. M. Rushby. *The PVS Proof Checker: A Reference Manual.* Computer Science Lab, SRI International.

Integrating Computational Properties at the Term Level

Martin Pollet[1]* and Volker Sorge[2]

[1] Fachbereich Informatik, Universität des Saarlandes, Germany,
pollet@ags.uni-sb.de, http://www.ags.uni-sb.de/{\homedir}pollet
[2] School of Computer Science, University of Birmingham, UK,
V.Sorge@cs.bham.ac.uk, http://www.cs.bham.ac.uk/{\homedir}vxs

1 Introduction

Human mathematicians often use representations for particular mathematical concepts that allow them to remember properties of the concepts. After its initial introduction or construction the further use of a concept abstracts from its construction. Moreover, special representations are used that are suitable for particular reasoning strategies, for example multiplication tables, matrices and diagrams [5].

On the contrary the representation of mathematical objects in deduction systems is often dictated by the requirements of the formalism and logic of a particular system. For instance, in lambda calculus sets are usually represented as lambda terms containing a disjunction of equalities. For example, a set of the form $\{a, b, c\}$ is represented as $\lambda x_\bullet (x{=}a \vee x{=}b \vee x{=}c)$. While these representations are theoretically suitable for reasoning about properties of the abstract mathematical concept they are often a hindrance when dealing with concrete objects in practice. The representations are often very different from the informal mathematical vernacular. Furthermore they are typically also less suited for direct computations and cannot be directly passed to a computer algebra system. This has the disadvantage that identification and interpretation of terms has to be implemented in the interface between deduction and computation.

We present the notion of annotated constants as an abstraction over the construction of concepts. It enables abstract, concise representation of a mathematical object together with implicit handling of its computational properties inside a theorem proving system. An annotated constant replaces the functional expression of an object such as a set or a list. It is treated as a constant of the formal language by the prover, but it is associated with a datastructure that contains a representation of the object, which is suitable input for special tactics or computer algebra systems. Moreover, annotated constants distinguish particular objects from regular constants; a fact that can be exploited for input and display purposes as well.

With annotated constants, trivial properties of concrete objects are already implemented on the term level. Moreover, they ease the detection of equality between objects and abstract from certain proof obligations needed to establish

* The author's work is supported by EU IHP grant Calculemus HPRN-CT-2000-00102.

necessary properties of the objects in question. Nevertheless annotated constants do not extend the formal language of the theorem prover as they can be expanded to their formal definitions on a more primitive term level and their required properties are rigorously checked.

We emphasise that annotated constants are a pragmatic approach to the representation of some concrete mathematical objects inside a theorem prover and are not a theoretical framework to encode semantic or heuristic informations such as existing formalisms which include annotations [4] or labels [2]. Moreover, their introduction does not extend the underlying formalism of the theorem prover as for instance the extension of type theory to inductive types does (cf. [7]). Finally, our approach aims at facilitating the application of external computer algebra systems inside the prover. Therefore, the objects are usually represented in a form that can be directly used as input for a computer algebra system, which, as a side-effect, also facilitates a human-oriented presentation of the objects. In particular, we do not intend to implement optimised datastructures for efficient computations (cf. [8]) from computer algebra itself.

2 Annotated Constants

Suppose we have a formal language $\mathcal{L}$, then an annotated constant is a triple $(k, \mathbf{a}, t)$, where k is a constant of $\mathcal{L}$, $\mathbf{a}$ is the annotation, and t is a term in $\mathcal{L}$ that is the formal definition of k.

The *annotation* $\mathbf{a}$ can be an arbitrary datastructure that may contain other terms (without free variables) of the language $\mathcal{L}$ and must fulfil the property that the *constant* k can be identified and the *defining term* t can be generated from the annotation. The datastructure for annotations is designed in such a way that particular relevant information about the objects becomes directly accessible, either to allow tactics to access their information, or to ease the communication with external systems, such as computer algebra systems. Annotations allow to identify different classes of mathematical objects not only for tactics but also to have a special display presentation.

The constant k is the formal representation and part of the language $\mathcal{L}$. With this representation we can map the properties of the annotations into the formal system, namely, that two annotations $\mathbf{a}$ and $\mathbf{a}'$ are equal if their associated constants are identical.

Annotated constants are introduced for numbers, lists, tuples, sets, and cycles. We describe the latter two in more detail.

Sets. Finite sets have a special notation in the mathematical vernacular, for example $\{a, b, c\}$. The information connected with this representations is, that it is a set, it contains finitely many elements, and the elements are explicitly given. Usually finite sets with different ordering of the elements, for example $\{a, b, c\}$ and $\{b, a, c\}$, are trivially identified. We tried to capture these properties with annotated constants for finite sets. The annotated constant allows to access the elements of a set without further analysis on lambda terms (the defining term for the constant) and already implements the equality for sets which differ only in the ordering of their elements.

Annotation for finite sets: The datastructure of sets (unordered lists), the elements of the sets are terms of the formal language, e.g., $\{b, a, c\}$ with $a, b, c \in \mathcal{L}$.

Constant: The identifier for the formal constant is generated from a duplicate free ordering of the elements of the set, for the example $k_{\{a,b,c\}} \in \mathcal{L}$.

Definition: The ordering of the elements of the set that is the annotation is also used to construct a lambda term as definition, e.g., $\lambda x.(x{=}a \vee x{=}b \vee x{=}c)$.

Cycles. A permutation is a bijective mapping of a finite set onto itself and is often given in cycle notation, for example, the permutation defined by the cycle (1 2 3) acting on the set $\{1, 2, 3\}$ maps 1 to 2, 2 to 3, and 3 to 1. The elements of a cycle have to be duplicate free, this property is verified during parsing and allows to detect mistakes already in the input specification (see Section 2.1). Annotated constants for cycles implement a basic equality on cycles, that is cycles are identified if their elements are only shifted, for examples, (1 2 3) and (2 3 1) are equal. For the definition of annotated constants for cycles we use a representation that is similar to the (input) format of the computer algebra system GAP.

Annotation for cycles: The datastructure of lists, when the elements are in $\mathcal{L}$ and of type integers. Additionally the list must be duplicate free, e.g., $(k_3\ k_1\ k_2)$ with constants $k_3, k_1, k_2 \in \mathcal{L}$ which are the constants of the annotations $3, 1, 2$ representing integers.

Constant: A constant representing the cycle in a normal form, that is, the minimal element of the cycle is shifted to the first position, e.g., $k_{(k_1\ k_2\ k_3)}{\in}\mathcal{L}$.

Definition: The term representing the shifted cycle constructed with $nil, cons \in \mathcal{L}$, e.g., $t_{(k_1\ k_2\ k_3)}{=}cons(k_1, cons(k_2, cons(k_3, nil)))$.

The definition of annotated constants includes the identification of the corresponding constant from its annotation. This identification lies outside of the formal system and can therefore be a possible source for errors. The verification of tactic applications which use the information provided by annotations, will be explained in Section 2.3.

2.1 Implementation of Annotated Constants

The basic functionality for handling annotated constants is implemented on the term level of the Omega system [6]. An annotated constant is essentially similar to regular constants: it has the datastructure it denotes as name and the same type as the expression (i.e. the defining term t) it represents. It can also be replaced by its defining term during the proof or when expanding the proof to check formal correctness. (The latter step is explained in more detail in Sec. 2.3.) Annotated constants do not have to be explicitly specified in the signature of the proof and their defining term is only computed when necessary.

We extended Omega's input language to provide markup for annotated constants to indicate the type of the object it represents. For each kind of annotated constant the term parser has to be extended by an additional function. This allows to specially parse annotations and to immediately transform them into a normal form representation. During parsing additional properties can immediately be checked and errors in the specification can be detected, for example that the cycle is duplicate free. In other words the check for pure syntactic correctness via type checking can be enriched by additional functionality to verify certain semantic properties of the input. An additional output function for each kind of annotated constant allows to have different display forms for presenting formulas to the user.

2.2 Manipulating Annotated Constants

Annotated constants are usually manipulated using specialised tactics, which can directly operate on the datastructures comprising the annotations. These datastructures are deliberately chosen to reflect the intuitive representation and to ease the communication with external systems by using their input representation as annotation.

The computations themselves are either implemented in the programming language underlying Omega or are performed using external computer algebra systems. For instance, functions on integers such as addition, multiplication, etc., are simply mapped to their counterpart in the Lisp programming language in which Omega is implemented. We have one tactic that simplifies expressions containing integers by executing the appropriate Lisp functions.

On the other hand operations on cycles such as their application or the composition of two cycles are executed using tactics that call the computer algebra system GAP [3]. The results are then directly incorporated into the proof. Since our notation of cycles is similar to the one used in GAP the translation is straightforward.

We use annotated constants in a case study in which they substantially contribute to the abstraction and simplification of proofs. The case study itself is concerned with the verification of computations on permutation groups performed by the computer algebra system GAP with the help of the proof planner of Omega. Permutations are formalised as sets of cycles and concrete permutations are represented by annotated constants. Due to lack of space we refer to [1] for details.

2.3 Guaranteeing Correctness

To guarantee correctness for a proof in Omega the tactics justifying single proof lines have to be expanded to a machine-checkable calculus level. Thereby it is often necessary that annotated constants are substituted by their defining terms, for instance, to verify single computational steps. Moreover, additional properties on the annotated constants need to be checked, if there are any. In particular, those properties that have been checked informally during parsing or generation of an annotated constant have to be painstakingly verified at the logic level.

For example, for cycles it is always crucial to verify that they are duplicate free lists of integers. This is achieved by recursively checking that a predicate *cycle* holds. For the concrete cycle (1 2 3) the first two steps of this verification are: $L_1 : cycle(k_{(1\ 2\ 3)})$, $L_2 : cycle(cons(1, cons(2, cons(3, nil))))$, $L_3 : 1 \notin \{2, 3\} \wedge cycle(cons(2, cons(3, nil)))$. In the first step the constant $k_{(1\ 2\ 3)}$ in line L_1 is replaced with its defining term $cons(1, cons(2, cons(3, nil)))$. The second step is the expansion of the cycle predicate in L_2 which yields two new proof obligations in L_3: to show that 1 is not an element of $\{2, 3\}$ and that (2 3) is again a valid cycle.

3 Conclusion and Future Work

With annotated constants we have introduced a technique to attach information to objects implemented as constants in the formal language. This extension does not change the formal system and has therefore no influence on the correctness. We have currently implemented annotations for some special types of mathematical objects (numbers, lists, sets, tuples, and cycles), however the extension for other special representations is straightforward. An interesting case are functions and operations which can be evaluated when they are applied to annotated constants. The operations of intersection and union could have computational information annotated to be used when they are applied to concrete sets. With functional annotations and arguments, one could think about annotations for terms, which are the result of the evaluation.

We tried to capture some aspects of mathematical representations with annotated constants. There are further issues to be considered, for example, how can different representations of the same concept be expressed. This is especially important when the key step of a proof is to use a suitable representation of the problem or if different representations are necessary for efficient computations. Naturally the use of various, possibly interchangeable, representations has substantial implications for the underlying prover and its formal system. Robustness of the proving techniques will need to be ensured by designing tactics, planning methods, appropriate matching algorithms, etc. that can handle different representations of the same mathematical object. Moreover, guaranteeing soundness, in particular for equality, will be less trivial than for annotated constants.

References

1. A. Cohen, S. H. Murray, M. Pollet, and V. Sorge. Certifying solutions to permutation group problems. *Proc. of CADE–19*, LNAI, 2003. Springer Verlag. to appear.
2. D. Gabbay. *Labelled Deductive Systems – Vol. 1.* Number 33 in Oxford Logic Guides. Oxford University Press, 1996.
3. The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.3*, 2002. `http://www.gap-system.org`.
4. D. Hutter. Annotated reasoning. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):183–222, 2000.
5. M. Kerber and M. Pollet. On the design of mathematical concepts. Technical Report CSRP-02-06, The University of Birmingham, School of Computer Science, 2002.

6. The Omega Group. Proof development with Omega. *Proc. of CADE–18*, vol. 2392 of *LNAI*, pages 143–148, 2002. Springer Verlag.
7. Coq Development Team. The coq proof assistant reference manual, 2002.
8. R. Zippel. *Effective Polynomial Computation*. Kluwer Academic Press, 1993.

Towards a higher reasoning level in formalized Homological Algebra*

Jesús Aransay[1], Clemens Ballarin[2], and Julio Rubio[1]

[1] Dpto. de Matemáticas y Computación. Univ. de La Rioja. 26004 Logroño (Spain).
{jesus-maria.aransay,julio.rubio}@dmc.unirioja.es
[2] Institut für Informatik. Technische Univ. München. D-85748 Garching (Germany).
ballarin@in.tum.de

Abstract. We present a possible solution to some problems to mechanize proofs in Homological Algebra: how to deal with partial functions in a logic of total functions and how to get a level of abstraction that allows the prover to work with morphisms in an equational way.

1 Introduction

EAT [8] and Kenzo [2] are software systems, written under Sergeraert's direction in the nineties, for Symbolic Computation in Algebraic Topology and Homological Algebra. They have achieved some results (homology groups) that have not been determined yet by any other means (neither theoretical nor computational ones). The systems are based on the intensive use of functional programming techniques, which enable in particular to encode and handle at runtime the infinite data structures appearing in Algebraic Topology algorithms.

In order to increase the reliability of the systems, a project to formally analyze fragments of the programs was undertaken. Some results related to the algebraic specification of data structures can be found in [4]. We are now interested in the algorithms dealing with the data structures; our goal is to give certified versions of some crucial fragments of Kenzo using the tactical theorem prover Isabelle [6]. Some previous works in the area of Group Theory and Algebraic Topology, useful libraries mainly for Group Theory and the expressiveness of higher-order logic were the reasons to choose Isabelle for our approach. ACL2 [3] might have been another possible environment, since it is written on Common Lisp (as Kenzo and EAT), but it is harder to deal in ACL2 with functional programming and higher order logic, both of them needed for our work.

The first result that we want to prove mechanically is the Basic Perturbation Lemma (BPL), since its proof has associated an algorithm which is used in Kenzo as one of the central parts of the program. Once we obtain a complete proof of the BPL, we also would like to get a certified version of the program from it. More information about the BPL can be found in [7]. In Section 2 we comment some of the proofs we have already obtained. In Section 3 we present the problems we have found trying to go further in the proof . In Section 4, we

* Partially supported by MCyT, project TIC2002-01626 and by CAR ACPI-2002/06

present the framework that we propose to overcome these difficulties. The paper ends with a conclusions section.

2 First lemma

In a previous article [1], a detailed proof of the BPL was given. This proof was based on a collection of seven lemmas which use mainly equational reasoning and algebraic structures, taking advantage of some special properties of chain complexes and graded group morphisms. The first lemma of this collection is:

Lemma 1. *Let* $(f, g, h) : D_* \Rightarrow C_*$ *be a chain complex reduction. Then, there exists a canonical and explicit chain complex isomorphism between* D_* *and the direct sum* $\mathrm{Ker}(gf) \oplus C_*$*. In particular,* $F : \mathrm{Im}(gf) \to C_*$ *and its opposite* $F^{-1} : C_* \to \mathrm{Im}(gf)$*, defined respectively by:* $F(x) := f(x)$ *and* $F^{-1}(x) := g(x)$*, are inverse isomorphisms of graded groups.*

Instead of using the complete definition of reduction, we selected some of its properties and then we proved in Isabelle a more generic lemma (the previous lemma is an immediate consequence of this):

Lemma 1′. *Let* C_* *and* D_* *be chain complexes, and* $f : D_* \to C_*$ *and* $g : C_* \to D_*$ *be chain complex morphisms such that* $fg = \mathrm{id}_{C_*}$*. Then* C_* *is isomorphic to* $\mathrm{Im}(gf)$*.*

Isabelle libraries specifying the needed algebraic structures were developed. The main difficulty was the very unnatural way of dealing with morphisms: instead of reasoning with them as "atomic" entities, we had to apply them to "generic" elements of their domains in order to simplify expressions inside of the algebraic structures. This task was even more complicated by the fact that several different domains (even for the same functional map) were involved in each step of the proof.

3 Second lemma

Once this lemma was proved in Isabelle, the following one to be studied was:

Lemma 2. *Let* D_* *be a chain complex,* $h : D_* \to D_*$ *(degree +1) a morphism of graded groups, satisfying* $hh = 0_{D_*}$ *and* $hd_{D_*}h = h$*. Let* p *be* $d_{D*}h + hd_{D_*}$ *and* $\mathrm{inc}_{\mathrm{Ker}(p)}$ *the canonical inclusion from* $\mathrm{Ker}(p)$ *to* D_**. Then* $(\mathrm{id}_{D_*} - p, \mathrm{inc}_{\mathrm{Ker}(p)}, h)$ *is a reduction from* D_* *to* $\mathrm{Ker}(p)$*.*

The same method used for the previous lemma could be also applied to this one, but some problems appeared. On the one hand, the size of the proof was too big to continue and the proof scripts were becoming more and more unnatural, which made it hard to follow the underlying idea of the proof. The reasons for this were again the use of partial functions within a logic of total functions (higher-order logic) and the lack of a tool to easily build new functions from the old ones in an equational way. On the other hand, several morphisms with different domains and codomains (as it can be seen in both lemma 1 and lemma 2) appeared, and this made many steps of the proof repetitive and complicated. It is almost sure that the lemma could be proved within this framework just by separating the necessary situations, but for this lemma (and even more for the following ones) at least 3 basic morphisms and 8 different combinations and restrictions of them were needed; the size of the proof would do it completely inaccesible. This is what made us look for a better framework.

4 New framework

The definition of equality for functions in Isabelle is taken from the *extensionality principle*: $f = g \Leftrightarrow \forall x. fx = gx$.

Obviously, this is valid for total functions, but in our case (and usually in computational mathematics) to deal with partiality in domains and codomains is necessary. When domains are restricted and non-total, this information is stored via the quantifiers, and simply a linear combination of morphisms can turn into something hard to understand (and hard to reason about in a mechanized way). Moreover, some trivial facts are hard to state and also to prove in a logic of total functions; for instance, the following one, which claims that for every function f:

$$f_{\mathrm{Ker}\, f} = 0_{\mathrm{Ker}\, f}$$

Another problem is the composition of morphisms. In Isabelle the usual composition is defined when functions are total on the types, but when restricted domains are used and a source set and a target set are declared, composition has to be compatible with these domains.

In order to satisfy this needs, our proposal is to define morphisms in a more generic sense storing information explicitly, instead of keeping it in the logical part. In order to work with the morphisms in an equational way, we also consider it necessary to keep the source and the target of the function. In Isabelle syntax, the generic type that we define for a morphism between chain complexes is:

record $('a,\ 'b)$ *MRP-type* $=$
  src $::$ $'a$ *chain-complex*
  trg $::$ $'b$ *chain-complex*
  map $::$ $'a \Rightarrow\ 'b$

This definition of the MRP-type is useful to work with composition of morphisms, since it allows direct access to the source and target structures and this is necessary to define *compositions* correctly from a mathematical point of view.

We also need a new equality definition to work with these triples. It should be more generic than the *extensionality principle*, in order to be able to prove identities like $f_{\mathrm{Ker}\, f} =_{equiv} 0_{\mathrm{Ker}\, f}$ in a direct way:

constdefs
equiv :: $[('a,\ 'b)\ \mathit{MRP\text{-}type},\ ('a,\ 'b)\ \mathit{MRP\text{-}type}] \Rightarrow \mathit{bool}$
equiv mrp1 mrp2 $\equiv$ $(\mathit{src\ mrp1} = \mathit{src\ mrp2}) \wedge (\mathit{trg\ mrp1} = \mathit{trg\ mrp2})$
$\wedge\ (\forall\, x \in \mathit{carrier}\ (\mathit{src\ mrp1}).\ \mathit{map\ mrp1}\ x = \mathit{map\ mrp2}\ x)$

Two morphisms are *equiv* whenever they have the same source and target, and produce the same result for all the elements of the source set. So, *equiv* is an equivalence relation between triples and once this has been proved, is possible to use it in Isabelle for equational reasoning. With this relation, the proof, for instance, of $f_{\mathrm{Ker}\, f} =_{equiv} 0_{\mathrm{Ker}\, f}$, becomes trivial. This implementation produces also a new level of abstraction, allowing to reason about functions without using elements of their domains.

Another important tool which would allow to reason more efficiently is the following lemma ($\langle , , \rangle$ denotes an object of MRP-type):

Lemma 3. Laureano's Lemma- *Let* $\langle g, C, D\rangle$ *and* $\langle f, A, B\rangle$ *be two morphisms between chain complexes satisfying* $\langle g, C, D\rangle \circ \langle f, A, B\rangle\, \mathrm{equiv} \langle h, A, D\rangle$ *and let* A' *be a subchain complex from* A*,* B' *a subchain complex from* C'*,* $\mathrm{Im}\, f$ *contained on* B'*, and* $\mathrm{Im}\, h$ *contained on* D'*. Then* $\langle g, C', D'\rangle \circ \langle f, A', B'\rangle\, \mathrm{equiv} \langle h, A', D'\rangle$.

This lemma gives a powerful tool for reasoning about morphisms that would be quite hard to prove in other environments. Of course, it is also valid to reason about any other type of algebraic structure (groups, rings, ...).

Up to now, we have only developed in Isabelle the definition of *equiv* and some of its basic properties and also some tools about compositions, but we have developed the proofs by hand and these two features have shown to be flexible enough to prove the lemmas we need. Most of the basic steps of the proofs depend only on equational reasoning (associativity, sum, inverse, ...) or in the property stated in the previous lemma. We do hope that these two tools will help us to finish the proof of the BPL much easier. Moreover, this implementation can be also useful for other works where reasoning about morphisms (and not about a concrete morphism or function) and their domains is needed and, of course, not only in Isabelle but also in other theorem provers.

5 Conclusions

We consider that the problem tackled, i.e., the verification of the Kenzo computer algebra system in Isabelle, is interesting and challenging and gives rise to new ideas on how to implement mathematical problems on an abstract level in a theorem prover. Our first approach using only the available features in Isabelle would have been possible, because it offers a complete implementation of higher-order logic, but due to its size it would not have been very useful, and not very readable. Both reasons led us to introduce tools to give a better mechanized proof of the BPL (and more generically, to reason about morphisms within any framework):

– A new definition for morphisms storing information about the domain and codomain, which helped to implement the basic operations (composition, addition, inverse, ...) and also a new equivalence relation between morphisms which allows us to compare partial functions directly in any logic of total functions.
– Some lemmas to reason about compositions where domain and codomain are modified and to state properties of morphisms starting from other morphisms, as we saw in lemma 3.
– A higher level of abstraction for equational reasoning on morphisms. The point is to prove in the formal system that the set of morphisms over a chain complex can be endowed with a ring structure and the morphisms between different chain complexes produce a group, and that both kind of morphisms can be composed within a generic framework.

Another solutions have been proposed to deal with partial functions in Isabelle (see, for instance, [5] where option types are used to encode partial functions) but a combination of the three tools we have just enumerated should be enough for reasoning in an abstract way in our problem.

Our aim is to implement in Isabelle these three points. We hope (and our attempts on paper prove it) that they will be useful to produce new lemmas in a quite readable way and also with a smaller size. We also think that this approach may be extended to other areas of mathematics where proving properties of functions is needed and also to other theorem provers which employ a logic of total functions.

References

1. J. Aransay, C. Ballarin and J. Rubio. *Deduction and Computation in Algebraic Topology.* In Proceedings IDEIA 2002, Universidad de Sevilla, pp. 47-54.
2. X. Dousson, F. Sergeraert and Y. Siret. *The Kenzo program.* `http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/`
3. M. Kaufmann, P. Manolios and J. Strother Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.
4. L. Lambán, V. Pascual and J. Rubio. *An object-oriented interpretation of the EAT system.* To appear in AAECC.
5. O. Müller and K. Slind. *Treating Partiality in a Logic of Total Functions.* The Computer Journal, 40(10): 640-652, 1997.
6. T. Nipkow, L. C. Paulson and M. Wenzel. *Isabelle/HOL: A proof assistant for higher order logic.* Lecture Notes in Computer Science, 2283, 2002.
7. J. Rubio and F. Sergeraert. *Constructive Algebraic Topology.* Lecture Notes Summer School in Fundamental Algebraic Topology, Institut Fourier, 1997.
8. J. Rubio, F. Sergeraert and Y. Siret. *EAT: Symbolic Software for Effective Homology Computation.* Institut Fourier, Grenoble, 1997.

Making proofs in a hierarchy of mathematical structures

A case study within the Computer Algebra System FoC

Virgile Prevosto[1,2] and Mathieu Jaume[1]

[1] SPI – LIP6 – University Paris 6
8 rue du Capitaine Scott, 75015, Paris, France
{Virgile.Prevosto,Mathieu.Jaume}@lip6.fr
[2] INRIA – Projet Moscova
B.P. 105 - F-78153 Le Chesnay, France

Abstract. The FoC language is dedicated to the development of certified computer algebra libraries. These libraries are based on a hierarchy of implementations of mathematical structures. A FoC structure, called a species, contains the declarations of primitive operations and axioms, the definitions of derived operations, and theorems with their proofs. It is built through inheritance steps from previously existing structures, which includes in particular the redefinition of an operation. In this paper, we show that such a redefinition may have an huge impact on some of the proofs of the species, and may even invalidate some of them. We also describe what information the FoC system can give to the user about the proofs he is doing. As a conclusion, we propose a coding style to minimize the number of proofs that must be redone after a redefinition.

1 Introduction – Motivation

Whereas proofs of algorithms or "small" programs are usually rather easy to obtain (or not overall difficult), they become more complicated when programming "in the large", since we have to ensure the coherence of the whole software architecture. This problem is particularly acute in the context of a hierarchical architecture, like in the B [1] approach involving refinements or in an object-oriented approach involving inheritance. Indeed, using such frameworks requires to perform analysis in order to check coherence properties (inheritance lookup, resolution of multiple-inheritance conflicts, dependency analysis, type-checking ...). Nevertheless, such paradigms provide powerful mechanisms that allow to introduce several concepts (definitions, specifications ...) in a very concise way and ease the reuse of previous developments. Unfortunately, few logical frameworks allowing to obtain formal proofs provide inheritance or refinements mechanisms.

In fact, when dealing with "standard" logical frameworks, like Coq [19], a proof assistant based on the calculus of inductive constructions [7], proofs are easier to obtain and executable code can be directly extracted from proofs [14]. However, making developments becomes a long run task leading to tedious and repetitious work since many definitions and properties have to be duplicated. Another drawback of this approach is that it is very difficult to make a prototype of a given program. On the contrary, it is very often the case that one has to make an huge proof before extracting the code. Furthermore, reusability becomes very difficult, especially for proofs (see [9]).

Inheritance and refinement mechanisms are powerful features which are particularly well-suited to develop a library of mathematical structures for a computer algebra system since they allow to make several refinements of a specification until providing an executable code. The computer algebra system FoC [16, 17, 4, 20] provides such features and also allows to express properties about the code and to make proofs. The computational part of the FoC library, mostly developed by R. Rioboo, implements mathematical structures up to multivariate polynomial rings and includes complex algorithms. In order to make proofs, the FoC compiler provides a completely formalized framework (in which the proof has to be done) taking into account all the hypotheses implicitly introduced by refinements or inheritance. Such a framework is obtained by a static analysis based on a complete axiomatization in Coq of inheritance and decl- and def-dependencies [3].

Due to the fact that a lot of proofs are needed, a problem of "proofware engineering" arises. Suppose, for example, you have given a specification of a GCD function, and you assert some properties on this function. Some of these properties may already be proved while others need a definition of the GCD function. The FoC language allows to do the proofs just when you decide, of course before using the unit being built. Is it a good idea to do the proofs as soon as possible? as late as possible? In others words, is there a notion of "right instant" to do proofs? We met this problem when training a group of twenty undergraduate students to do "little" proofs in FoC. In this paper, we first describe this problem in detail and we make some propositions to avoid it as much as possible in a FoC development. Note that this problem is quite general and the issues presented here may be useful in other systems.

2 A brief description of the computer algebra system FoC

The FoC project[3] aims at building an environment to develop certified computer algebra libraries. In the FoC language, any implementation must come with a proof of its correctness. This includes of course pre- and post- condition statements, but also proofs of purely mathematical theorems. In this context, *reusability* of code, but also of proofs is very important: a tool written for the groups should be available for the rings, provided the system knows that every ring is a group. Thus, like Axiom, FoC is based on a hierarchy of mathematical structures.

Since we want to do some proofs, we need also a completely formalized representation of the inheritance relations between the different structures of the

[3] `http://www-spi.lip6.fr/~foc`

hierarchy. This formalization points out that some invariants must be preserved when extending an existing structure. We have then elaborated a concrete syntax, allowing the user to write programs, statements, and proofs. It is restrictive enough to prevent some inconsistencies but not all. So the FoC compiler must make some analyzes to check the correctness of FoC programs. They are then translated into our target languages, that is

– OCAML, a functional language of the ML-family, which can be compiled in very efficient native code.
– COQ, a proof-checker which offers powerful constructions and is quite close to OCAML.

FoC's Ground Concepts

Species. Species are the nodes of the hierarchy of structures that makes up the library. They correspond to the algebraic structures in mathematics. A species can be seen as a set of *methods*, which are identified by their names. In particular, there is a special method, called the *carrier*, which is the type of the representation of the underlying set of the algebraic structure. It is represented by the keyword **self**. Every method can be either *declared* or *defined*.

– *Declared* methods introduce the primitive constants and operations of the structure. Moreover, axioms are also represented by declared methods, as would be expected by the Curry-Howard isomorphism.
– *Defined* methods represent implementations of operations and proofs of theorems.

Currently, a proof in FoC consists in a COQ script interpreted in the context of the species where it is done. Some syntactic sugar has been added to tell the FoC compiler what are the *dependencies* (see below) of such a proof, so that the compiler can set up the appropriate environment when translating a FoC program into COQ.

Collection. A *collection* is built upon a completely defined species. This means that every method must be defined. In other words, in a collection, every operation has an implementation, and every theorem is formally proved. In addition, a collection is "frozen". Namely, it cannot be used as a parent of a species in the inheritance graph, and its carrier is considered as an abstract data type. A collection represents an implementation of a particular mathematical structure, such as $(\mathbb{Z}, +, *)$ implemented upon the GMP library. Note that the user of a collection does not know the representation of the carrier and thus, cannot break representation invariants.

Def- and Decl- Dependencies A method m_1 of a given species s can call a method m_2 of s through the use of a method call (**self**!m_2 in the FoC syntax). We then say that m_1 *depends upon* m_2. Dependencies may arise in various places:

– in the definition of a function,

– in the statement of a property or a theorem, or
– in the proof of a theorem.

In the latter case, we must distinguish between two kinds of dependencies:

– there is a *def-dependency* upon m_2 if we need to know the exact *def*inition of m_2 to do the proof[4]
– there is a *decl-dependency* otherwise (we only need to know the *decl*aration of m_2, *i.e.* its type or its statement).

A more formal definition of def- and decl- dependencies, can be found in [18]. To clarify the difference between def- and decl- dependencies, we can take the example of a setoid. The species has the following declared methods: a carrier `rep`, an abstract equality `eq`, and three properties stating that `eq` is reflexive, symmetric and transitive. From `eq`, we define its negation `neq`, and prove, by the theorem `neq_nrefl`, that it is irreflexive. In FoC, we represent such a `setoid` like this (some properties and the proof script have been omitted):

```
species setoid =
 rep;
 sig eq in self -> self -> bool;
 let neq (x,y) = #notb(self!eq(x,y));
 property eq_refl: all x in self, self!eq(x,x);
 theorem neq_nrefl: all x in self, not(self!neq(x,x))
    proof: def neq;
           decl eq_refl;
           ... ;
end
```

If we observe the body of `neq`, we remark that it def-depends upon `eq`. Similarly, `eq_refl` depends upon `eq`, which is needed to express the statement of the property. The statement of `neq_nrefl` depends upon `neq`, but if we want to do the proof of `neq_nrefl`, it is not sufficient to know the type of `neq`. Indeed, we have to know that `neq` is the boolean negation of `eq`, so that we can apply the reflexivity of `eq` to conclude. This means that there is a *def*-dependency upon `neq` (but also a *decl*-dependency upon `eq_refl`).

When dealing with inheritance, *def-dependencies* have a major drawback: if we want to redefine `neq` in a species that inherits from `setoid`, then we will have to provide a new proof for `neq_nrefl`, since the one we have here relies on the old definition of `neq` given in `setoid`.

3 Erasure of Proofs during Inheritance: a Complete Example

In this section, we present a complete example illustrating the problem that may happen during the redefinitions. We then give two issues to this problem.

[4] Note that proof-irrelevance leads to a system in which there is no def-dependencies upon theorems.

In the FoC hierarchy, the species of partially ordered sets inherits of setoids and introduces a partial order relation `leq` together with its properties of reflexivity, antisymmetry and transitivity. From these declared methods, a strict order relation `lt` is defined and a theorem expressing the relations between `leq`, `lt`, `eq` and `neq` is proved (here again, we omit some properties and proofs):

```
species partial_order inherits setoid =
sig leq in self->self->bool;
let lt(x,y) = #and_b(self!leq(x,y),#not_b(self!eq(x,y)));
theorem lt_is_not_leq : all x y in self,
  ((self!lt(x,y) -> (self!leq(x,y) and self!neq(x,y))) and
  (self!leq(x,y) -> (self!lt(x,y) or self!eq(x,y))))
  proof: def: lt neq;
         decl: leq;
          ... ;
...
end
```

The proof of theorem `lt_is_not_leq` depends upon `lt` and `neq`. Now, note that this species introduces `lt` directly by a definition without giving any of its properties. The relation `lt` is coded but not specified, thus the only possibility is that `lt_is_not_leq` def-depends upon `lt` and `neq`.

Going a step further, from partially ordered sets, by adding a property, we can define totally ordered sets, thus allowing us to use `leq` to give a definition to `eq`. Hence, we are now able to prove the reflexivity, symmetry and transitivity properties of `eq`. Such proofs def-depend upon the definition of equality. From this new property, we can also redefine the relation `lt` from `leq`.

```
species ordered_set inherits partial_order =
 property total_order : all x y in self,
  #or_b(self!leq(x,y),self!leq(y,x));
 let eq(x,y) = #and_b(self!leq(x,y),self!leq(y,x));
 proof of equal_reflexive = decl: leq_reflexive;
                            def: equal;
                            ...;
 let lt(x,y) = #not_b(self!leq(y,x));
 proof of lt_is_not_leq = ... ;
 ...
end
```

Now, since the theorem `lt_is_not_leq` proved in the species `partial_order` def-depends upon the “old” definition of `lt`, we have to prove it again according to the new definition of `lt`. Indeed, due to the management of such def-dependencies by the compiler, the current proof has been fortunately erased during inheritance.

Now, depending on the context of development, we can wonder about the usefulness of the proof of `lt_is_not_leq` in the species `partial_order`. Of course, if many species or collections inherit of partial_order, proving lt_is_not_leq in the species partial_order allows to avoid many duplications (since the first definition of lt is often used). However, in the FoC hierarchy, no species (except ordered set) or collections inherit of partial_order, and then such a proof is not really necessary since the first definition of lt is never used. In fact, it is always the redefinition of lt which is used in the numerous species or collections which inherit from ordered_set since they use total orderings. But further developments of the library may use partial orderings.

What are the issues to this question? The first one is to give a true specification of lt, that is a declaration of lt and a property characterizing it. This "specification by properties" approach is the one used in the B system. It is also reminiscent of the definition of partial functions by equations where these equations are described by a predicate that can be directly translated in a Prolog style (see [8]). However, in these approaches, the computational part is not considered. With this approach, partial_order would be written that way:

```
species partial_order inherits setoid =
 sig leq in self->self->bool;
 sig lt in self->self->bool;
 property spec_lt : all x y in self,
  !lt(x,y) <-> (!leq(x,y) and !neq(x,y));
 theorem lt_is_not_leq : all x y in self,
   ((!lt(x,y) -> (!leq(x,y) and !neq(x,y))) and
   (!leq(x,y) -> (!lt(x,y) or !eq(x,y))))
   proof: def:neq;
          decl: leq spec_lt;
           ...;
...
end
```

Remark: A complete application of this solution would have rewritten setoid as well to give a specification neq_spec of neq, so that lt_is_not_leq would not def-depend upon neq anymore.

Now, the proof of lt_is_not_leq has only a decl-dependency upon lt, thus it is insensitive to (multiple) inheritance or late-binding. Note that the first definition of lt is very close to the specification spec_lt. Even if these two approaches seem similar, they correspond to two different points of view. Later, at each (re)definition of lt, as in the species ordered_set for example, we will have to prove the property spec_lt.

The second issue is closer to an implementation. It may be useful when the first definition of lt was already given and used, properties and proofs being added later. So starting from the definition of lt, we introduce spec_lt as a new theorem. Then we are able to prove, as in the preceding solution, lt_is_not_leq without using the definition of lt, by using only a decl-dependency upon spec_lt. This property can be seen as the "minimal hypothesis" on lt which is required to prove lt_is_not_leq. Hence, during inheritance, instead of (re)proving `lt_is_not_leq`, it suffices to (re)prove `is_lt` (which should be shorter than the proof of `lt_is_not_leq`). In other words, the property `is_lt` seems to be the "good cut".

```
species partial_order inherits setoid =
 sig leq in self->self->bool;
 let lt(x,y) = basics#and_b(!leq(x,y),basics#not_b(!eq(x,y)));
 theorem spec_lt : all x y in self,
  !lt(x,y) <-> (!leq(x,y) and !neq(x,y))
 proof :def : lt neq;
       decl: leq;
       ...;
 theorem lt_is_not_leq : all x y in self,
  ((!lt(x,y) -> (!leq(x,y) and !neq(x,y))) and
  (!leq(x,y) -> (!lt(x,y) or !eq(x,y))))
  proof: def:neq;
        decl:spec_lt leq;
        ...;
...
end
```

Remark: Once again, it would be possible to have a theorem `spec_neq` in `setoid`, so that `lt_is_not_leq` would not have any def-dependency anymore.

In this case, if one defines a species or a collection which inherits of `partial_order` without redefining `lt`, he can use the proof of `spec_lt` given in `partial_order`.

4 Adopting a New Coding Style to Make the Proofs

We can now try to analyze this example a bit further in order to state rules that may ease the development of proofs in such an object-oriented framework. As we have already noticed, there are two important questions.

- The first one is to find a set of properties such that the redefinition of the specified function invalidates as few proofs as possible.
- The second one is to decide at which level of the hierarchy a proof has to be done.

At the end of the section, we also discuss how the FoC system can help in making such choices.

As a preliminary step, we assume that the typical development process of a FoC library is often made of two steps. First, there is the elaboration of the hierarchy of species with the computational methods attached to each of them, and the specifications of the functions. Then, even if it is not recommended, the proofs are usually done after the code has been tested (this allows to check CPU-time or memory neededed). This is a way to avoid trying to prove erroneous implementations. However, proofs that depend only on specifications can be done before any test.

4.1 Expressing Specifications

Before doing any proof, the first thing to do is to express the properties that have to be verified by the methods of a species. From the example above, we can draw the following guidelines to write such properties:

- even if we do not make the proof formally, we should have in mind what are its def- and decl- dependencies.
- there should be as few theorems with def-dependencies as possible
- theorems with def-dependencies have to be simple, so as to minimize the work to be done in case of redefinition
- a given theorem should have at most one def-dependency. Otherwise, it might indicate that a function has been underspecified and that we have to rely on its definition instead of its specification. In other words, you have to convince yourself that there is no way to escape from a def-dependency.

4.2 When Should we do the Proofs ?

The main lesson of the example above, is that it is quite difficult to find the "right instant" when to do a proof. More precisely, if one wants to prove a property P in a given hierarchy of species, one must take into account two things:

1. the framework of each species of the hierarchy. In particular, one has to know which functions are defined and which properties are available in a given species.
2. the global inheritance graph. Here, the number of child(ren) of each species can be very important. Indeed, as we said, if this number is big, it may be a good choice to do proofs as soon as possible, while if this number is rather small, it may be better to delay the proofs.

By inspecting the local context of each species, we can find the first species s which is refined enough in order to prove the theorem. But as said before, it does not mean that we *prove* P in this species s: if P def-depends upon a method x and x is redefined in every child of s, then the proof given in s will never be used in any implementation. In this case, it may be better to delay the proof of P, even if it could be done already in s. On the other hand, if the definition of x found in s persists until a collection is implemented, then P should be proved in s.

4.3 A Little Help from the Compiler

When the hierarchy of structures is large, it might become difficult to track by hand every inheritance step in order to find out where a particular proof has to be done. To face this problem, some tools are provided by the FoC system in order to trace dependencies. They take advantage of the analyzes performed by the FoC compiler to show some informations that may help the user in this choice.

First, a warning is issued each time a proof of a property P is erased due to the redefinition of a function x. In addition, the name of the species s_1 where P is proved as well as the name of the species s_2 where x is redefined are reported. If this warning is issued too many times for the same P and x, it may indicate that the proof of P came too early in the inheritance graph and that s_2 might be a better choice.

Second, FoC can tell the user that in a species s, all the functions involved in the statement of a property P are defined. Actually, some experiences showed that most of the time, if a proof def-depends upon x, x appears in the statement of the theorem. Thus this warning may be a sign that the species s is a good place to prove P.

5 Related work

Algebraic hierarchies have been developed in various theorem-provers or proof-checkers. In particular, Loïc Pottier [15] has developed an huge library in Coq about fundamental notions of algebra, up to fields. H. Geuvers and the FTA project [11] are also using the `Records` of Coq to represent algebraic structures, in order to define abstract and concrete representations of reals and complex numbers. Similarly, the Mizar Project has built since 1989 a fairly large database of important theorems of mathematics in the Mizar Mathematical Library [22]. However, none of these works involve a computational counterpart, as in FoC. In particular, this means that there is no redefinition issues in these hierarchies: once a theorem has been proved, its proof remains correct in the following structures. The need of replacing a proof by another one only arises when one wants to replace a generic algorithm by a more specialized (and more efficient one).

On another side, several attempts have been made to interface an existing computer algebra system (CAS) and a theorem prover or a proof assistant. The so-called "Skeptic's approach" of Harrison and Théry [12] consists in verifying a result given by a CAS (for instance the primitive of a given function computed by Maple) with a Theorem Prover (in their example, by formally deriving the result in HOL). In other words, instead of proving the correctness of an algorithm, they simply ensure the correctness of each returned result. This approach is much lighter than FoC, since it relies on existing an CAS. However, it can be done only when the verification of the result is much simpler than the computation (deriving vs. integrating).

Similarly, Dunstan, Gottliebsen Kelsey, and Martin [10] designed and interface between Maple and PVS which allowed them to extend Maple with the proof features of PVS. In particular this interface can correct some errors such as a simplification made without paying attention to the validity of the input (such as $\sqrt{x^2} \rightarrow x$, which Maple usually performs whether x is positive or not). Such an approach is very interesting since it ensures that Maple procedures are called in a suitable context. But given the lack of semantics of Maple's language it seems rather difficult – if not impossible – to prove the correctness of Maple procedures as we could attempt to do it in FoC.

Another way to provide certified computer algebra programs is to use the extraction facilities provided by Coq. This has been done by Théry [21] on the Buchberger algorithm. He first proved the correctness of this algorithm in Coq. Then he used the extraction mechanism to get an OCAML implementation of the algorithm from his proof. While such an approach guarantees that the extracted program is correct, it does not allow a "prototyping" phase. Indeed, one must do the entire proof before extracting any useful program. When dealing with new algorithms, that may be still under design, one might want to adopt a softer approach, which, like FoC, allows to test the implantation (its correctness, but also its efficiency) before having all the proofs done.

In addition, we must mention the TH∃OREM∀ [6] system, which is implemented in Mathematica. It consists in several provers able to handle various proof situations. TH∃OREM∀ attempts to provide an unified framework in which the user can specify its problem, make some proofs, but also write some algorithms which can help him to solve the initial problem. When programming such an algorithm, the user of the system can also of course take advantage of the computational power of Mathematica. However, it seems to be quite different from FoC in the sense that it does not rely on a hierarchy of mathematical structures as the species and collections of FoC.

Last, the notion of development graphs used in the MAYA system [2] seems to be closely related to the problem of proof erasure during inheritance. Indeed, development graphs are used to minimize the number of proofs that are to be done when the specification of a given system changes during the development (possibly to add a new functionality or to fix a bug). However, there are still some differences. In fact MAYA is mainly dedicated to study the propagation of a change in an structure through an already existing hierarchy. On the contrary, we have tried here to address the issue of creating a new structure from existing one through inheritance step. In other words, MAYA operates on a closed hierarchy in which nodes may evolve, while FoC uses an open hierarchy with fixed nodes.

6 Perspectives

The current stage of the FoC library contains only a few proofs with respect to the computational part. One of the main tasks that remain to do is to complete the specifications of each of the functions used in the library, following the methodology of section 4.1. Then, according to section 4.2 we will have to decide where these specifications should be proved, and make the proof.

The last step is likely to be quite long and tedious, especially if it is done with Coq scripts. Actually, such a script is most of the time done interactively, which means that in order to do a proof, we must first compile all the species into Coq with a dummy proof, find the place which corresponds to the theorem in the resulting file, write the script by interacting with the Coq interpreter, and copy-paste it in the original FoC file. It is clearly impossible to do a large amount of proofs that way, and we are currently designing our own proof tool. It is based on Lamport's hierarchical style [13], in which a proof consists of intermediate lemmas, each of them being proved by other lemmas, until we reach some trivial properties. One of the interests of this style is that it might allows us to refine the def-dependency analysis, so that a redefinition might not invalidate a whole proof, but only some of the intermediate lemmas used in it.

Another point which could help us writing proofs would be of course to reuse as much as possible the existing ones, especially those who have been done in COQ. A first attempt has been done in [9], which showed on an example from the group theory that this was possible, but a lot of work is needed to obtain a systematic way to reuse such proofs.

7 Conclusion

In this paper, we showed how classical inheritance mechanisms of object-oriented programming are not harmless when dealing with theorems. The main point here is that the redefinition of a function during inheritance may invalidate some proofs, which are then erased and must be redone in the new context.

We also proposed a methodology to try to minimize the impact of such a redefinition, that is to invalidate as few proofs as possible, and to invalidate small proofs rather than long and complicated ones. Thanks to that, we hope to be able to ease the development of the certified part of the FOC library. Indeed, when including proofs in the hierarchy, a methodological question arises: when or where must we include the proofs? As we said, the naive answer to this question consisting in making proofs as soon as possible might not always be the good answer, and is definitely not sufficient as a coding style.

In the long run, the FOC compiler might evolve into a system in which the user could easily specify its algorithms, make some prototypes, and prove the correctness of the implementations once they are mature. In this context, the hierarchical structure of the FOC libraries would also allow to build such programs step by step, and to have a very fine control over each part of the algorithms, by choosing the exact place where to do each proof.

References

1. J.R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. S. Autexier and D. Hutter. Maintenance of formal software development by stratified verification. In *Proceedings of LPAR*, volume 2514 of *LNCS*, Tbilisi, Georgia, October 2002. Springer.
3. S. Boulmé. *Spécification d'un environnement dédié à la programmation certifiée de bibliothèques de Calcul Formel.* PhD thesis, Université Paris 6, 2000.
4. S. Boulmé, Th. Hardin, D. Hirschkoff, V. Ménissier-Morain, and R. Rioboo. On the way to certify computer algebra systems. In *Proceedings of the Calculemus workshop of FLOC'99 (Federated Logic Conference, Trento, Italy)*, volume 23 of *ENTCS*. Elsevier, 1999.
5. B. Buchberger. Symbolic computation: Computer algebra and logic. In F. Baader and K.E. Schultz, editors, *Proceedings of the "Frontiers of Combining Systems" conference*, Applied Logic Series. Kluwer, 1996.

6. B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuta, and D. Vasaru. A survey on the theorema project. In W. Kuechlin, editor, *Proceedings of ISSAC'97*. ACM Press, 1997.
7. T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2-3):95–120, 1988.
8. C. Dubois and V. Viguié Donzeau-Gouge. A step towards the mechanization of partial functions: domains as inductive predicates. In *Workshop on mechanization of partial functions, CADE-15*, 1998.
9. C. Dubois, J. Grandguillot, and M. Jaume. Réutilisation de preuves formelles : Une étude pour le système FoC. In INRIA, editor, *14ème Journées Francophones des Langages Applicatifs, JFLA'2003*, pages 63–75, 2003.
10. M. Dunstan, H. Gottliebsen, T. Kelsey, and U. Martin. Computer algebra meets automated theorem proving: A maple-pvs interface. In *Proceedings of the Calculemus Workshop*, 2001.
11. H. Geuvers, R. Pollack, F. Wiedijk, and J. Zwanenburg. The algebraic hierarchy of the fta project. In *Proceedings of the Calculemus Workshop*, 2001.
12. J. Harrison and L. Théry. A skeptic's approach to combining HOL and Maple. *Journal of Automated Reasoning*, 21:279–294, 1998.
13. L. Lamport. How to write a proof. research report, Digital Equipments Corporation, February 1993.
14. C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15(5–6):607–640, 1993.
15. L. Pottier. contrib algebra pour coq, March 1999. `http://\*pauillac.inria.fr/\*coq/\*contribs-eng.html`.
16. V. Prevosto and D. Doligez. Algorithms and proof inheritance in the foc language. *Journal of Automated Reasoning*, 29(3-4):337–363, dec 2002.
17. V. Prevosto, D. Doligez, and Th. Hardin. Algebraic structures and dependent records. In S. Tahar C. Munoz and V. Carreno, editors, *TPHOLs : 15th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, volume 2410. LNCS, Springer-Verlag, August 2002.
18. S. Ranise. Combining generic and domain specific reasoning by using contexts. In Jacques Calmet et al., editors, *Proceedings of AISC-Calculemus*, volume 2385 of *LNAI*, pages 305–318, Marseille, July 2002. Springer-Verlag.
19. The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 7*. INRIA-Rocquencourt, 2002.
20. The FoC Development Team. *The FoC System Reference Manual Version 0.0*. LIP6 – INRIA – CNAM, 2003.
21. L. Théry. A certified version of buchberger's algorithm. In *Proceedings of CADE-15*, pages 349–364, Lindau, Germany, July 1998.
22. A. Trybulec et al. Mizar mathematical library. `http://mizar.uwb.edu.pl/library/`.

Formal proofs and computations in finite precision arithmetic

Sylvie Boldo[1], Marc Daumas[1], and Laurent Théry[2]

[1] Laboratoire de l'Informatique du Parallélisme
UMR 5668 CNRS – ENS de Lyon – INRIA
Lyon, France
Sylvie.Boldo@ENS-Lyon.Fr & Marc.Daumas@ENS-Lyon.Fr

[2] Dipartimento di Informatica
Università di L'Aquila
L'Aquila, Italy
Laurent.Thery@di.univaq.it

Abstract. In this paper, we give examples of application in the specific area of floating point arithmetic where being able to mix formal reasoning and computing power is mandatory.

1 Introduction

Hardware units and algorithms manipulating floating point numbers are applications where formal proofs and in particular theorem proving have demonstrated to be useful. Pioneering works such as [9, 20, 29, 34] have shown both the benefits one could get from using formal methods and some pitfalls that should be avoided [3]. Inspired by these works, we have been developing our own library to reason about floating point numbers in the Coq system [25]. This library has been built as generic as possible. Properties are proved in the most general possible setting, and only when necessary a particular radix, a particular format, or a particular rounding mode is introduced.

Using our library, not only already known results have been formally proved for the first time but also new results have been established [7, 10]. In that respect using a theorem prover has been a stimulating and productive experiment. While trying to apply our library to more and more complex examples, we were quickly limited by the simple fact that Coq was really bad at computing. The applications we are considering really require a system good at doing *both* deduction *and* computation.

In this paper we present some of these applications and propose them as challenges for systems claiming to integrate deduction and computation. The paper is organized as follows. In Section 2 we give a short introduction on IEEE floating point numbers. In Section 3 we present some of our results and explain how having some basic computations could have largely improved our productivity. Section 4 and Section 5 have a more prospective flavour and describe some applications we have just started looking at. Section 4 describes two general methods for validating approximations of elementary and special functions.

Section 5 gives examples where validating the actual code manipulating floating point numbers used by computer algebra systems could be of some interest.

2 Floating point numbers

Floating point numbers are defined by the IEEE-754 standard [37]. They are represented using bits s, $e_{l-1}\cdots e_0$ and $d_{-1}\cdots d_{1-\text{prec}}$ interpreted as shown below:

$$\begin{cases} (-1)^s \cdot \left(1 + \displaystyle\sum_{i=-1}^{1-\text{prec}} d_i 2^i\right) \cdot 2^{\sum_{i=0}^{l-1} e_i 2^i + 1 - 2^{l-1}} & \text{if some } e_i\text{'s are not equal to zero} \\ (-1)^s \cdot \left(\displaystyle\sum_{i=-1}^{1-\text{prec}} d_i 2^i\right) \cdot 2^{2-2^{l-1}} & \text{if all } e_i\text{'s are equal to zero} \end{cases}$$

In our formalization we do not use the exact representation of the hardware floating point numbers defined by the IEEE-754. Instead, we define a generic floating point number as a pair of integers (n, e). They are mapped from $\mathbb{Z}^2$ onto $\mathbb{R}$ by

$$(n, e) \hookrightarrow n\beta^e$$

where β, the radix of the floating point system, is an integer constant strictly greater than one. We focus our interest on pairs (n, e) such that n and e are bounded in order to be representable as a floating point pair. This means

$$|n| < \beta^{\text{prec}} \qquad \text{and} \qquad e \geq -e_{min}$$

with prec > 1.

Given a number p, the weight of the last bit/digit of the IEEE-like mantissa is called $ulp(p)$. The rounded value of a real value r is denoted $\circ(r)$ and must be one of the two floats that are around it. When the real to be rounded can be represented by a bounded floating point number, the two floating point numbers around it are purposely equal. The standard defines 4 rounding modes (round to nearest, round up, round down, round to zero) that determine uniquely what the rounded value of any real value is.

Following the philosophy of the IEEE standard, all operations on floating point numbers should return the rounded value of the result of the exact operation. For example, the addition of two floating point numbers a, b should always return $\circ(a + b)$ where $+$ is the usual addition over the reals. Sometimes correct rounding is impossible to ensure and we just have *faithfulness*. An operations is faithful if the result is either the rounding up or the rounding down of the real exact value.

When performing a sequence of floating point operations, each single operation may introduce a rounding error. Thus, the result can be arbitrary far from the exact value. The absolute value of the difference between the result and the exact value is called the *error*.

3 Manipulating inequalities explicitly

Proofs in computer arithmetic are mainly used to formally establish bounds on errors. To illustrate what a proof in computer arithmetic is, consider the theorem `RoundLeGeneral` used in [5]. The theorem `RoundLeGeneral` states that, if p is a representable floating point number that is a rounding to the nearest of a real z, then

$$|p| \leq \frac{|z|}{1-\beta^{-\mathrm{prec}}} + \frac{\beta^{-e_{min}-1}}{1-\beta^{-\mathrm{prec}}}$$

The beginning of the pen and paper proof looks like this:

$$\begin{array}{lr@{}l}
 & |p| & {}\leq \frac{|z|}{1-\beta^{-\mathrm{prec}}} + \frac{\beta^{-e_{min}-1}}{1-\beta^{-\mathrm{prec}}} \\
\Leftrightarrow & (1-\beta^{-\mathrm{prec}}) \times |p| & {}\leq |z| + \beta^{-e_{min}-1} \\
\Leftrightarrow & |p| - \left(|p| \times \beta^{-\mathrm{prec}} + \beta^{-e_{min}-1}\right) & {}\leq |z| \\
\Leftrightarrow & |p| - \left(|p| \times \beta^{-\mathrm{prec}} + \beta^{-e_{min}-1}\right) & {}\leq |p| - \frac{\mathrm{ulp}(p)}{\beta} \quad \text{and} \quad |p| - \frac{\mathrm{ulp}(p)}{\beta} \leq |z|
\end{array}$$

The first two steps are simplifications and the third one splits the inequality into two subproblems. Unfortunately the interaction with the prover requires many more steps:

$$\begin{aligned}
|p| &\leq \frac{|z|}{1-\beta^{-\mathrm{prec}}} + \frac{\beta^{-e_{min}-1}}{1-\beta^{-\mathrm{prec}}} \\
\left(1-\beta^{-\mathrm{prec}}\right) \times |p| &\leq \left(1-\beta^{-\mathrm{prec}}\right) \times \left(\frac{|z|}{1-\beta^{-\mathrm{prec}}} + \frac{\beta^{-e_{min}-1}}{1-\beta^{-\mathrm{prec}}}\right) \\
|p| - |p| \times \beta^{-\mathrm{prec}} &\leq \left(1-\beta^{-\mathrm{prec}}\right) \times \left(\frac{|z|}{1-\beta^{-\mathrm{prec}}} + \frac{\beta^{-e_{min}-1}}{1-\beta^{-\mathrm{prec}}}\right) \\
|p| - |p| \times \beta^{-\mathrm{prec}} &\leq |z| \times \frac{1-\beta^{-\mathrm{prec}}}{1-\beta^{-\mathrm{prec}}} + \beta^{-e_{min}-1} \times \frac{1-\beta^{-\mathrm{prec}}}{1-\beta^{-\mathrm{prec}}} \\
|p| - |p| \times \beta^{-\mathrm{prec}} &\leq |z| \times 1 + \beta^{-e_{min}-1} \times 1 \\
|p| - |p| \times \beta^{-\mathrm{prec}} &\leq |z| + \beta^{-e_{min}-1} \\
-\beta^{-e_{min}-1} + \left(|p| - |p| \times \beta^{-\mathrm{prec}}\right) &\leq -\beta^{-e_{min}-1} + \left(|z| + \beta^{-e_{min}-1}\right) \\
-\beta^{-e_{min}-1} + \left(|p| - |p| \times \beta^{-\mathrm{prec}}\right) &\leq |z| \\
|p| - \left(|p| \times \beta^{-\mathrm{prec}} + \beta^{-e_{min}-1}\right) &\leq |z| \\
|p| - \left(|p| \times \beta^{-\mathrm{prec}} + \beta^{-e_{min}-1}\right) &\leq |p| - \frac{\mathrm{ulp}(p)}{\beta} \quad \text{and} \quad |p| - \frac{\mathrm{ulp}(p)}{\beta} \leq |z|
\end{aligned}$$

Note that the first and fourth steps also require to prove that the quantity $0 < 1 - \beta^{-\mathrm{prec}}$ is strictly positive. This adds an extra four steps to the proof:

$$\begin{aligned}
0 &< 1 - \beta^{-\mathrm{prec}} \\
\beta^{-\mathrm{prec}} + 0 &< \beta^{-\mathrm{prec}} + (1 - \beta^{-\mathrm{prec}}) \\
\beta^{-\mathrm{prec}} &< \beta^{-\mathrm{prec}} + (1 - \beta^{-\mathrm{prec}}) \\
\beta^{-\mathrm{prec}} &< 1 \\
\beta^{-\mathrm{prec}} &< \beta^0 \quad \text{and} \quad 1 = \beta^0
\end{aligned}$$

The three steps of the paper and pencil proof are now fourteen steps in the prover.

Another example is the computation of $a \times x + y$. The real computation is $\circ(\circ(a \times x) + y)$ with two rounding errors. In [5] it has been proved that this computation is faithful under few assumptions:

Theorem 1 (Axpy_opt) *Given real numbers a_0, x_0 and y_0, and bounded floating point numbers a, x and y, if no overflow occurs and if*

$$5\,\frac{2+\mathrm{ulp}}{2-\mathrm{ulp}} \times \left(|a \times x| + \frac{\lambda \times \mathrm{ulp}}{2}\right) \leq |y| \quad \textit{and}$$

$$|y_0 - y| + |a_0 \times x_0 - a \times x| \leq \frac{\mathrm{ulp}}{8}\left((1-\mathrm{ulp}) \times |y| - |a \times x| - 2 \times \lambda\right),$$

then $\circ(\circ(a \times x) + y)$ is a faithful approximation of $a_0 \times x_0 + y_0$.

where λ is the smallest normal positive number and ulp is an abbreviation for *ulp*(1). For sake of simplicity, we ask that $\mathrm{ulp} \leq 2^{-3}$. That condition will be met by every reasonable hardware implementation.

The proofs works as follows. We first distinguish two cases. First, if $r \leq f$ we prove that $|f - r| < \mathrm{ulp}(f^-)$ where f^- is the float predecessor of f. Second, if $r \leq f$ we prove that $|f - r| < \mathrm{ulp}(f)$. These two subgoals are then proved using a fair amount of simplifications but only using few basic properties of the roundings. Moreover, having to apply manually the simplification steps make it difficult to factorize subproofs. For example, the proof is further split into three subcases. We note $t = \circ(a \times x)$ and $u = \circ(\circ(a \times x) + y) = \circ(t + y)$.

- When $t + y \leq u$, after some work we are left to prove that
$$|t + y - u| \leq \frac{1}{2}\mathrm{ulp}(u^-).$$
- When $u < t + y \leq u + \frac{1}{2}\mathrm{ulp}(u^-)$, after some work we are left to prove the (easy) fact that
$$|t + y - u| \leq \frac{1}{2}\mathrm{ulp}(u^-).$$
- When $u + \frac{1}{2}\mathrm{ulp}(u^-) < t + y$, after some work, we are left to prove that
$$|t + y - u| \leq \mathrm{ulp}(u) - \frac{1}{2}\mathrm{ulp}(u^-).$$

All the proofs follow the same general pattern but each subcase has its own details. It follows that the Coq file is 3 400 lines long and about 80 % of it is only dealing with equalities/inequalities.

What makes the formal proofs so long is that we have to explicitly guide the prover for performing simplifications. Simplifying expressions automatically is possible, all computer algebra systems do that. Automatic simplification would yield a much more comfortable and productive interaction with the proof system.

We have been developing dedicated tactics to perform some basic simplifications. With respect to what is proposed by computer algebra systems (see for example [31]), our simplifications try to avoid as much as possible to modify the overall structure of the formula. The idea is to get as close as possible to simplifications that are done in paper and pencil proofs. For example given the formula

$$x(z + y + 2) < y(3x + 1)$$

all the simplification can do is to modify the coefficient of the different monomials

$$\boxed{1}x(\boxed{1}z + \boxed{1}y + \boxed{2}) < \boxed{1}y(\boxed{3}x + \boxed{1}).$$

A possible simplification is

$$\boxed{1}x(\boxed{1}z + \boxed{\mathbf{0}}y + \boxed{2}) < \boxed{1}y(\boxed{\mathbf{2}}x + \boxed{1})$$

giving the result

$$x(z + 2) < y(2x + 1)$$

We still need to further develop our set of tactics so to make our library an effective environment to formally check the kind of proofs we have presented here.

4 Deriving inequalities from function analysis

For a few decades, many implementations have been proposed for the elementary functions [32] and a few ones for the special functions. These functions cannot be implemented exactly in a finite number of steps. They have to be approximated. Straightforward approximations such as Taylor expansions are quite slow and many improvements have been used to enhance speed and precision of the algorithms. It seems impossible to avoid small mistakes such as the one presented in [19] about one seminal publication [38]. As even small errors may trigger catastrophic results, libraries should be checked with some formal tool.

Lately, designers favour the use of polynomial approximations of the target function over a set of small intervals, i.e. the function $f(x)$ is approximated by a polynomial $P(X)$ over an interval I. Below, we consider two of the options available for establishing and checking the error bounds of such approximations. The first one has already been successfully used to validate an implementation of the exponential function. It requires some non-trivial computations (Taylor's expansion, isolating roots). The second one is even more challenging since computation is at the heart of the validating process.

4.1 Root isolation on the derivative of the error function.

The coefficients of $P(X)$ are floating point numbers (rational numbers) and have been computed elsewhere. Formal proof checking is only applied to validate the error bounds $f(x)-P(x)$. As stated in [19], this "procedure has the added advantage that we can take from the literature the polynomial approximations actually considered by other workers, without worrying about whether they coincide with the best approximations as we would calculate them."

The error bound can be checked by isolating all the zeros of $f'(x) - P'(x)$. Isolation intervals for the different roots are reduced and $f(x)-P(x)$ is bounded on each interval containing a root of $f'(x) - P'(x)$. Algebraic tools have long been used to isolate zeros of polynomials, but they do not work on polynomials mixed with transcendental functions.

Harrison [19] replaced the target function $f(x)$ by a large-degree Taylor's expansion $T(X)$ that is very close to $f(x)$. The analysis is then carried on using the well-known Sturm's series to isolate the zeros of the polynomial $P'(x)-T'(x)$. The final bound is $||f(x) - T(x)||_\infty + ||T(x) - P(x)||_\infty$ where the first norm is bounded by one of Taylor's formulas.

4.2 Interval arithmetic and bisection.

Many authors [18, 27] have proposed to use interval arithmetic to find guaranteed extrema of functions. Given an expression g or a program that does not use branches, we can compute a superset of $g(J)$ for any interval J. This could be achieved by validating formally a program that evaluates the expression g using a validated library for interval arithmetic.

It is well known that interval analysis usually yields pessimistic bounds. There is no chance that computing $f(x) - P(x)$ on interval I itself would lead to any usable error bound on P but this process can be refined by replacing I by sub-intervals I_k such that $I \in \bigcup I_k$. Recursively sub-intervals I_k can be split down to the point where all interval supersets $f(I_k)$ are sufficiently accurate.

This is how a more realistic maximum of a function can be computed: we handle a set of intervals (I_k) and a lower bound on the maximum of f.

- $(I_k) \leftarrow I$
  $f_{max} \leftarrow -\infty$
- Each interval I_k is examined: we compute the interval evaluation X of f over I_k, then:
  - If $X_{max} \leq f_{max}$, it guarantees that the maximum is not in I_k and I_k is discarded.
  - Otherwise if I_k is the smallest interval allowed (between a floating point number and its successor in the given format), then the maximum can be computed on this interval with a realistic bound.
  - Otherwise the interval I_k is split in two intervals (lower and upper half) that are added to the (I_k). The interval I_k itself is discarded. The maximum is updated to X_{min} if $f_{max} \leq X_{min}$.

From a theoretical point of view, this algorithm should work, but a rule of thumb predicts that we will need as many as 2^{prec} subintervals to obtain a precise upper bound on the error.

To get better results, we use a property of differentiable functions over a compact set: the extrema are located either on the border of the compact set or on a root of its gradient. If the function can be differentiated twice, we can also test if its Hessian matrix is symmetric positive and definite, thus meaning that the zero of the gradient is not located on a saddle point. Testing the matrix may be long, so we usually test a weaker condition. For example, we check that all the diagonal components are positive. If any of these tests fails, the interval is discarded.

For the computation of the maximum, it means that if the interval evaluation of the gradient of f does not contain 0 or if its Hessian matrix is symmetric positive (or something weaker), the interval I_k can be discarded.

In practice, this code is easy to parallelize and program can be significantly sped up by using multiple processors and/or machines over a network.

5 Aggregating little rounding errors into intervals

Floating point arithmetic has been used by computer algebra systems for different reasons.

5.1 Description and analysis of numerical programs

To describe faithfully the behaviour of a numerical program using floating point arithmetic, computer algebra systems should include a model of floating point arithmetic. The model should be very close to the machine implementation and should have been studied in details. These are two of the goals of our library [10, 6].

Melquiond [28] has implemented and studied a program to detect future intrusions of airplanes into conflict areas. This work is based on an earlier implementation [14] computing on ideal real numbers and supposing that the airplanes evolve in a Euclidean 3D space. The algorithm has been thoroughly tested and its proof of correctness was validated using PVS.

Melquiond took care of floating point round-off errors and of the fact that airplanes are localised and move over a spheroid (the earth). Small errors are scattered along the path of the development of this new algorithm. Both geometric and round-off errors are easily defined and handled using a computer algebra system because elaborate mathematics is easily used. Yet the proof should be validated using an automatic proof checker.

5.2 Approximated solutions

Numerical linear toolboxes such as MATLAB or MAPLE (NAG's library) heavily use fast floating point arithmetic to compute approximated solutions [23, 2]. Mathematical problems are studied through backward analysis independently of the implemented algorithm. Quantities such as condition numbers are defined for some of these problems. The condition number is estimated by the algorithm that computes the approximated solution [12, 24].

Validating such developments would need an automatic proof checker able to handle many results of mathematics. Such developments are becoming available [16]. Using them for numerical linear algebra would be both a full scale test and a strong achievement to be promoted to people working in scientific computing and numerical analysis. Such techniques are still applied to develop new algorithms robust to small errors and to characterise the effect of rounding errors on existing ones.

5.3 Filters

Some libraries use adaptative operators that are able to estimate how accurate is the temporary result and can refine it to a higher accuracy and/or precision if needed. Some of these libraries, including [36], heavily use the properties on round-off errors of floating point operations.

The round-off errors of the addition [30, 26] and of the multiplication [11] can be represented by a floating point number, and an algorithm is available to compute it exactly using only common arithmetic operations. We can compute in a similar way the remainder of the division or the square root [4]. These properties have long been used and have been recently validated using Coq [7]. As the round-off error of each intermediate operation is computed, it is easy to compensate for it or estimate its effect on the temporary result [41].

From a practical point of view, it seems that we should not refine any temporary result. The very first step would use fast floating point arithmetic to solve all the easy problems. The other problems are handled exactly using the multiple precision numbers available in computer algebra systems. We can establish a law, related to Amdahl's law, since it was proved for many examples that easy problems appear much more frequently [13, 17].

We are currently working on Horner's rule. This rule appears in transformations applied in Descartes rule [33] used in computer algebra systems to isolate roots of polynomials. It is also used in floating point libraries to accurately evaluate polynomial approximations of the elementary and special functions [5].

6 Conclusion

The examples we gave in this paper show that computer arithmetic is an area where mixing deduction and computation would find natural applications. Following the pioneering work by John Harrison [22], provers like Coq, Isabelle, Pvs are actively formalising real analysis and start covering most of the mathematics needed for the applications that we have described here.

The ways to get computation and deduction in a single system are well known. The most pragmatic approach is to link together an existing theorem prover and an existing computer algebra as in [1]. From the software engineering point of view, the maintenance of such a system should not be underestimated. Furthermore, keeping the combined system consistent is problematic.

A drastic approach proposed in [21] is to make the prover always check the result of the computer algebra system. In some particular situations, checking could be far simpler than computing. It is not clear if such an approach would be sufficient for the applications we are considering. Provers like Coq provide limited amount of compilation that is an order of magnitude smaller than what is available in a programming language like C. For example, in [39] we have evaluated that what Coq could reasonably compute amounts to the equivalent of 1 second of execution time of the corresponding ML program performing the same computation.

Of course all these problems would not exist if deduction and computation were built in the system from the beginning. Interesting approaches follow this line such as [8] but they do not seem mature enough to tackle the applications we are interested in. A promising alternative is the one proposed in [35]. The idea is to build a system doing computation and deduction not as a single engine but as a set of little engines, each engine performing a very precise task. This of course would only work if the implementations of the various engines were done with lots of care. They should act like reference implementations where both efficiency and correctness were taken into account. Ultimately, it could also be possible to formally prove the correctness of these implementations, as it has already been done, for example, for a specific algorithm of computer algebra [40]. We are currently working on this aspect, trying to extend the verification condition generator Why [15] for Coq so to handle programs manipulating floating point numbers.

References

1. Andrew Adams, Martin Dunstan, Hanne Gottliebsen, Tom Kelsey, Ursula Martins, and Sam Owre. Computer Algebra Meets Automated Theorem Proving: Integrating MAPLE and PVS. In *TPHOLs'01*, volume 2152 of *LNCS*, pages 27–42, Edinburgh, Scotland, 2001.
2. E. Anderson et al. *LAPACK users' guide*. Society for Industrial and Applied Mathematics, 1995. Second edition.
3. Geoff Barrett. Practical algorithms for selection on coarse grained parallel computers. *IEEE Transactions on Software Engineering*, 15(5):611–621, 1989.
4. Gerd Bohlender, Wolfgang Walter, Peter Kornerup, and David W. Matula. Semantics for exact floating point operations. In Peter Kornerup and David Matula, editors, *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 22–26, Grenoble, France, 1991.
5. Sylvie Boldo and Marc Daumas. Faithful rounding without fused multiply and accumulate. In *IMACS-GAMM International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*, Paris, France, 2002.
6. Sylvie Boldo and Marc Daumas. Properties of the subtraction valid for any floating point system. In *the 7th International Workshop on Formal Methods for Industrial Critical Systems*, pages 137–149, Málaga, Spain, 2002.

7. Sylvie Boldo and Marc Daumas. Representable correcting terms for possibly underflowing floating point operations. In Jean-Claude Bajard and Michael Schulte, editors, *Proceedings of the 16th Symposium on Computer Arithmetic*, Santiago de Compostela, Spain, 2003.
8. Sylvain Boulmé, Therèse Hardin, and Renaud Rioboo. Some hints for polynomial in the FOC project. In *Calculemus 2001 Proceedings*, 2001.
9. Victor A. Carreño and Paul S. Miner. Specification of the IEEE-854 floating-point standard in HOL and PVS. In *1995 International Workshop on Higher Order Logic Theorem Proving and its Applications*, Aspen Grove, Utah, 1995. Supplemental Proceedings.
10. Marc Daumas, Laurence Rideau, and Laurent Théry. A generic library of floating-point numbers and its application to exact computing. In *14th International Conference on Theorem Proving in Higher Order Logics*, number 2152 in LNCS, pages 169–184, Edinburgh, Scotland, 2001.
11. Theodorus J. Dekker. A floating point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.
12. James W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.
13. Olivier Devillers and Franco Preparata. A probabilistic analysis of the power of arithmetic filters. *Discrete and Computational Geometry*, 20(4):523–547, 1998.
14. Gilles Dowek, César Muñoz, and Alfons Geser. Tactical conflict detection and resolution in a 3-d airspace. In *Proceedings of the Fourth International Air Traffic Management R&D Seminar ATM-2001*, 2001.
15. Jean-Christophe Filliâtre. Proof of Imperative Programs in Type Theory. In *TYPES '98*, number 1657 in LNCS, Eindhoven, Netherlands, 1998.
16. Herman Geuvers, Freek Wiedijk, and Jan Zwanenburg. A constructive proof of the fundamental theorem of algebra without using the rationals. In *TYPES'00*, number 2277 in LNCS, pages 96–111, 2000.
17. Richard W. Hamming. The unreasonable effectiveness of mathematics. *American Mathematical Monthly*, 87(2):81–90, 1980.
18. Eldon Hansen, editor. *Global optimization using interval analysis*. Marcel Dekker publisher, 1992.
19. John Harrison. Floating point verification in HOL light: the exponential function. Technical Report 428, University of Cambridge Computer Laboratory, 1997.
20. John Harrison. A machine-checked theory of floating point arithmetic. In *the 12th International Conference on Theorem Proving in Higher Order Logics*, number 1690 in LNCS, pages 113–130, Nice, France, 1999.
21. John Harrison and Laurent Théry. A Skeptic's Approach to Combining HOL and Maple. *Journal of Automated Reasoning*, 21(3):279–294, 1998.
22. John R. Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998.
23. Desmond J. Higham and Nicholas J. Higham. *MATLAB Guide*. SIAM, 2000.
24. Nicholas J. Higham. *Accuracy and stability of numerical algorithms*. SIAM, 2002. Second edition.
25. Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The Coq proof assistant: a tutorial: version 7.2. Technical Report 256, Institut National de Recherche en Informatique et en Automatique, Le Chesnay, France, 2002.
26. William Kahan. Further remarks on reducing truncation errors. *Communications of the ACM*, 8(1):40, 1965.
27. R. Baker Kearfott, editor. *Rigorous global search: continuous problems*. Kluwer Academic Publishers, 1996.
28. Guillaume Melquiond. Robustesse d'algorithmes d'évitement des collisions aériennes. Master's thesis, École Normale Supérieure de Lyon, Lyon, France, 2003.

29. Paul S. Miner and James F. Leathrum. Verification of IEEE compliant subtractive division algorithms. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, pages 64–78, 1996.
30. Ole Møller. Quasi double-precision in floating point addition. *BIT*, 5(1):37–50, 1965.
31. Joel Moses. Algebraic simplification a guide for the perplexed. In *Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*, pages 282–304, 1971.
32. Jean-Michel Muller. *Elementary functions, algorithms and implementation*. Birkhauser, 1997.
33. Renaud Rioboo. *Programmer le Calcul Formel, des Algorithmes à la Sémantique*. Habilitation à diriger des recherches, Université Pierre et Marie Curie, Paris, France, 2002.
34. David M. Russinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998.
35. Natarajan Shankar. Little engines of proof. In *LICS*, Copenhagen, Denmark, 2002.
36. Jonathan R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. In *Discrete and Computational Geometry*, volume 18, pages 305–363, 1997.
37. David Stevenson et al. An American national standard: IEEE standard for binary floating point arithmetic. *ACM SIGPLAN Notices*, 22(2):9–25, 1987.
38. Ping Tak Peter Tang. Table driven implementation of the exponential function in IEEE floating point arithmetic. *ACM Transactions on Mathematical Software*, 15(2):144–157, 1989.
39. Laurent Théry. Stålmarck's Algorithm in Coq: A Three-Level Approach. Technical Report 4353, INRIA, 2002.
40. Laurent Théry. A certified version of Buchberger's algorithm. In *Automated Deduction—CADE-15*, volume 1421 of *LNAI*, pages 349–364. Springer-Verlag, 1998.
41. J. H. Wilkinson. *The algebraic eigenvalue problem*. Oxford University Press, 1965.

Inductive definitions versus classical dependent choice in the Minlog system

Ulrich Berger and Monika Seisenberger
{u.berger,csmona}@swansea.ac.uk

University of Wales Swansea

1 Introduction

Minlog, developed by the group of Helmut Schwichtenberg at the University of Munich, is an interactive proof system based on Heyting Arithmetic in finite types that allows for extracting programs from constructive and classical proofs [BBS+98,BSS01]. This paper is concerned with extensions of this mechanism by program extraction from proofs involving constructive inductive definitions on the one hand and classical choice principles on the other. Both extensions will be discussed by means of an extended case study whereby special emphasis is put on obtaining optimized extracted code: we distinguish between computationally relevant and irrelevant predicates and quantifiers and allow the external animation of higher order constants using side effects to improve efficiency.

The case study is about Higman's Lemma, a result in infinitary combinatorics which is used in term rewriting theory to prove termination of string rewriting systems. We implemented two versions: one uses Nash-Williams' classical 'minimal bad sequence argument', the other is based on constructive inductive definitions but uses the combinatorial idea implicit in Nash-Williams' proof. While the inductive proof admits a rather straightforward realizability interpretation via infinitary algebraic data types, the problem of extracting computational content from Nash-Williams' proof hinges on a translation of classical into constructive proofs (A-translation) and on a suitable constructive interpretation of the A-translated principle of classical dependent choice [BO03,BBC98]. For an overview on Higman's Lemma we refer to [Fri97] or [Sei03]. The inductive version of Higman's Lemma was first proven in [CF94]; a generalization of this proof to an arbitrary well-quasiordered alphabet may be found in [Sei01]. The first implementation of Nash-Williams' proof is due to [Mur90], cf. section 4.

For later reference we recall the statement of Higman's Lemma and its classical proof due to Nash-Williams [NW63] whereby, for simplicity, we restrict ourselves to a two letter alphabet $A = \{0, 1\}$. Let A^* be the set of finite sequences in A, also called words. A word v is *embeddable* in w ($v \leq^* w$) if v can be obtained from w by deleting some letters. A (finite or infinite) sequence of words $(w_i)_{i<I}, I \leq \omega$, is *good* if there are indices $i < j < I$ such that $w_i \leq^* w_j$; otherwise it is called *bad*.

Higman's Lemma Every infinite sequence of words is good.[1]

Proof (Nash-Williams,1963). Assume for contradiction that there is an infinite bad sequence of words. Then, we define a 'minimal' bad sequence as follows: If $w_0, \ldots, w_{n-1}$ are already chosen, we choose w_n minimal with respect to the initial segment relation such that $w_0, \ldots, w_{n-1}, w_n$ can be extended to an infinite bad sequence of words. Clearly, the resulting infinite sequence $(w_n)_{n<\omega}$ is bad. Therefore all w_n are nonempty and can be written as $w_n = v_n * a_n$, that is, a_n is the last letter of w_n. In the sequence $(a_n)_{n<\omega}$ either 0 or 1 occurs infinitely often, hence there is a strictly increasing sequence of indices $(n_i)_{i<\omega}$ such that $a_{n_0} = a_{n_1} = \cdots$. Now, the infinite sequence $w_0, \ldots, w_{n_0-1}, v_{n_0}, v_{n_1}, \ldots$ is bad and contradicts the definition of the minimal bad sequence.

2 Inductive definitions

From a constructive point of view it is important that the notion of a well-quasiorder can be formulated without referring to infinite sequences using a (generalized) inductive definition instead. In the following we sketch the inductive formulation of Higman's Lemma and use it to explain inductive definitions and their constructive interpretation via infinitary algebraic types in Minlog. To this end we inductively define a set $\mathsf{Bar} \subseteq A^{**}$ by

$$\frac{\mathsf{Good}\, ws}{\mathsf{Bar}\, ws} \qquad\qquad \frac{\forall w\, \mathsf{Bar}\, ws * w}{\mathsf{Bar}\, ws}$$

where we use the notation ws for a finite sequence of words $[w_0, \ldots, w_{n-1}]$. Classically, $\mathsf{Bar}\, ws$ holds if every infinite sequence starting with ws is good.

Inductive formulation of Higman's Lemma $\mathsf{Bar}\,[\,]$

Using the induction principle for the predicate Bar, we may prove that the inductive formulation of Higman's Lemma implies the classical one. The converse can be shown by a non-constructive choice principle which will be discussed in detail in the next section.

The inductive proof of Higman's Lemma is carried out in intuitionistic arithmetic plus induction over inductive predicates and inductive types.

The computational content of an inductive definition On the program side an inductive definition corresponds to an inductively defined data type whose constructors are determined by the 'types' of the closure axioms. Intuitively, this type consists of wellfounded trees indicating how elements entered the inductively defined set. The induction principle for an inductive definition is realized by recursion on the corresponding inductive type.

[1] Higman's Lemma in its general form is phrased "If $(A, \leq_A)$ is a well-quasiorder, then so is $(A^*, \leq_{A^*})$" where for a quasi order $(A, \leq_A)$ being a well-quasiorder means that every infinite sequence in A is good.

Optimizations We allow inductive definitions having no computational content, i.e., no type is assigned to them. Examples are the inductive characterizations of the embeddability relation

$$\frac{}{[] \leq^* []} \qquad \frac{v \leq^* w}{v \leq^* w * a} \qquad \frac{v \leq^* w}{v * a \leq^* w * a}.$$

and the predicate Good. Furthermore, we distinguish between two types of quantifiers, the usual quantifier $\forall$ and a quantifier $\forall^{\mathsf{nc}}$ which has the same logical behavior, but carries no computational content. In our example, this has the effect that, for instance, the first closure axiom can be reformulated as $\forall^{\mathsf{nc}} ws\, (\mathsf{Good}\, ws \to \mathsf{Bar}\, ws)$ which corresponds to the simple constructor $\mathsf{Leaf} : \mathsf{tree}$.

3 Classical Dependent Choice

Now we show how to extract computational content directly from Nash-Williams' proof. The essential computational ingredient in the extracted program will be a form of recursion over wellfounded trees given by continuous type 2 functionals (in contrast to wellfounded trees as elements of an inductive type).

Gödel- and A-translation of classical proofs Recall that in Nash-Williams' proof one derives a contradiction from the assumption that a given sequence f of words is bad. We use Gödel's negative translation combined with the Friedman/Dragalin A-translation to transform this classical proof into a constructive proof. For convenience we will in the following call this translation simply A-translation although we mean in fact the combination of Gödel- and A-translation. In its original form the A-translation amounts to double negating all atomic and existential formulas (where $\neg B$ is defined as $B \to \bot$) and replacing $\bot$ (falsity) by the formula A stating that f is good, i.e., $A :\equiv \exists i < j\, f(i) \leq^* f(j)$. Under this translation all axioms concerning classical logic become intuitionistically provable, and instances of mathematical principles like induction are translated into (different) instances of the same principle. Most importantly, the (false) assumption that f is bad is translated into an intuitionistically provable formula. Altogether one obtains an intuitionistic proof of the translation of $\bot$ from which a program computing indices i, j with $f(i) \leq^* f(j)$ can be extracted.

The translation of dependent choice When applying the A-translation to Nash-Williams' proof one encounters a problem, namely, the definition of the minimal bad sequence. In order to carry out this definition formally one needs an axiom of the form

$$\textbf{DCseq} \quad B([]) \wedge \forall ws (B(ws) \to \exists w\, B(ws * w)) \to \exists g \forall n\, B(\bar{g}n)$$

where $\bar{g}n := [g(0), \dots, g(n-1)]$. This is easily proven to be intuitionistically equivalent to the more common scheme of dependent choice

$$\textbf{DC} \quad \forall n \forall x \exists y\, B(n,x,y) \to \forall x \exists f\, (f(0) = x \wedge \forall n\, B(n, f(n), f(n+1))).$$

A-translation of **DC** leads to a formula $\mathbf{DC}^A$ which is *not* an instance of dependent choice nor can be intuitionistically proven from it. Nevertheless it is possible to give $\mathbf{DC}^A$ (and hence $\mathbf{DCseq}^A$) a constructive interpretation. A realizer of $\mathbf{DC}^A$ can be constructed, i.e., (Gödel) primitive recursively defined from the following recursively defined functional

$$\textbf{BR} \quad \Psi(Y, H, xs) = Y(\lambda k.\mathsf{if}\ k < |xs|\ \mathsf{then}\ xs_k\ \mathsf{else}\ H(xs, \lambda x.\Psi(Y, H, xs * x)))$$

where xs varies over finite sequences of some type ρ and the equation is of ground type ([BO03]; an interpretation of $\mathbf{DC}^A$ by Ψ appears first in [BBC98]).

Optimizations *1. Simplifying types and control.* As explained above the (unrefined) A-translation replaces every atomic formula C by $(C \to A) \to A$ where A is an existential formula. This has the effect that higher types and many case distinctions come up in the extracted program. In [BBS02] a refined A-translation is introduced that allows to minimize double negations and hence reduce these negative effects. These refinements are implemented in Minlog and have been tested in our case study.

2. Improving efficiency. In Minlog the functional Ψ can be introduced as a program constant together with a rewrite rule corresponding to **BR**. When running (i.e. normalizing) programs containing Ψ one observes however a certain inefficiency which can be explained by the fact that when (in **BR**) Y calls its argument at different values $\geq |xs|$ the expression $H(xs, \lambda x.\Psi(Y, H, xs{*}x))$ (which does not depend on k) is evaluated repeatedly. An obvious method to avoid this inefficiency is to equip the argument of Y with an internal memory that stores the value of the expression $H(xs, \lambda x.\Psi(Y, H, xs{*}x))$ after being computed the first time. It is possible to do this in Minlog because Minlog implements normalization by evaluation [BES98]: Instead of introducing a rewrite rule one animates the constant Ψ by a (Scheme) procedure that accomplishes the desired memoization via a side effect.

4 Conclusion

In this paper we discussed two versions of Higman's Lemma and their extracted programs (for the implementation see `www.minlog-system.de`). In both cases the main computational principle used is recursion on wellfounded trees. However, whereas in the inductive proof the trees are inductively generated as the elements of an inductive data-type, in Nash-Williams' proof no new data-types need to be introduced and wellfounded trees are given by the unsecured sequences of a total continuous functional. Although both forms of wellfounded recursion are known to be of different strength in general ([Spe62], [Tro73], Appendix by J. Zucker), it is possible that the particular instances used here are in some way equivalent. It was our hope that by analyzing the extracted programs such an equivalence could be revealed. Unfortunately, the program corresponding to the second version is still too complex to permit such an analysis, although it is considerably shorter than the program extracted in [Mur90]. It also remains unclear how our programs are related to those extracted in [Mur90] and [Her94].[2]

Furthermore, we would like to remark that the inductive approach presented in this paper may be carried out in any theorem prover supporting program extraction from inductive definitions.[3] The second approach is more specific to the Minlog system, since it requires an implementation of the refined A-translation which does not seem to be available in other systems. In particular, the optimizations via memoization directly rely on Minlog's normalization procedure.

In a future project the refined A-translation could also be applied to the classical proof of Kruskal's Theorem, even in its strong form with gap condition [Sim85]. The latter would be interesting because then we could extract a program from a theorem for which no constructive proof is known so far.

References

[BBS+98] H. Benl, U. Berger, H. Schwichtenberg, M. Seisenberger, and W. Zuber. Proof theory at work: Program development in the Minlog system. In W. Bibel and P. H. Schmitt, editors, *Automated Deduction – A Basis for Applications*, volume II, pages 41–71. Kluwer, Dordrecht, 1998.

[BBC98] S. Berardi, M. Bezem, and T. Coquand. On the computational content of the axiom of choice. *The Journal of Symbolic Logic*, 63(2):600–622, 1998.

[BBS02] U. Berger, W. Buchholz, and H. Schwichtenberg. Refined program extraction from classical proofs. *APAL*, 114:3–25, 2002.

[BES98] U. Berger, M. Eberl, and H. Schwichtenberg. Normalization by evaluation. In B. Möller and J.V. Tucker, editors, *Prospects for Hardware Foundations*, LNCS 1546, pages 117–137. Springer, 1998.

[BO03] U. Berger and P. Oliva. Modified Barrecursion and Classical Dependent Choice. 2003. To appear in Lecture Notes in Logic, 19 pages.

[BSS01] U. Berger, H. Schwichtenberg, and M. Seisenberger. The Warshall Algorithm and Dickson's Lemma: Two Examples of Realistic Program Extraction. *Journal of Automated Reasoning*, 26:205–221, 2001.

[Ber03] S. Berghofer. *Proofs, Programs and Executable Specifications in Higher Order Logic*. PhD thesis, Technical University Munich, 2003. Forthcoming.

[CF94] T. Coquand and D. Fridlender. A proof of Higman's lemma by structural induction, 1994. ftp://ftp.cs.chalmers.se/pub/users/coquand/open1.ps.Z.

[Fri97] D. Fridlender. *Higman's Lemma in Type Theory*. PhD thesis, Chalmers University of Technology and University of Göteburg, 1997.

[Her94] H. Herbelin. A program from an A-translated impredicative proof of Higman's Lemma. http://coq.inria.fr/contribs/higman.html, 1994.

[Mur90] C. R. Murthy. *Extracting Constructive Content from Classical Proofs*. PhD thesis, Ithaca, New York, 1990.

[2] [Her94] comprises a program obtained from the A-translated proof in Coq. In [Mur90] and [Her94] infinite sequences are formalized as graphs (given by predicates) and extracted programs are second-order polymorphic.

[3] See e.g. [Ber03] for an implementation of Higman's Lemma in Isabelle and [Fri97] for a formalization of a different proof in Alf.

[NW63] C. St. J. A. Nash-Williams. On well–quasi–ordering finite trees. *Proc. Cambridge Phil. Soc.*, 59:833–835, 1963.

[Sei01] M. Seisenberger. An Inductive Version of Nash-Williams' Minimal-Bad-Sequence Argument for Higman's Lemma. In P. Callaghan, e.al., *Types for Proofs and Programs*, LNCS 2277, Springer, 2001.

[Sei03] M. Seisenberger. *On the Constructive Content of Proofs*. PhD thesis, University of Munich, 2003. Forthcoming.

[Sim85] S. G. Simpson. Nonprovability of certain combinatorial properties of finite trees. In L.A. Harrington, e.al., *Harvey Friedman's Research on the Foundations of Mathematics*, pages 87–117. North–Holland, 1985.

[Spe62] C. Spector. Provably recursive functionals of analysis: a consistency proof of analysis by an extension of principles in current intuitionistic mathmatics. In F. D. E. Dekker, *Recursive function theory*, 1–27. North–Holland, 1962.

[Tro73] A. S. Troelstra. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, LMS 344, Springer Verlag, 1973.

Building Convex Hulls by Combining SAT Solving and Algebraic Computing

Silvio Ranise

LORIA & INRIA-Lorraine
Silvio.Ranise@loria.fr

Abstract. Geometric algorithms (such as those for computing convex hulls) are quite difficult to be implemented correctly and efficiently. This is so because of errors deriving from arithmetic operations or of contradictory results returned by primitives operations. In this abstract, we present a combination of SAT solving and algebraic computing to flexibly build parsimonious algorithms, which are both correct and efficient. The algorithm can be lifted to handle the case in which input data are described by means of arithmetic constraints. Finally, we present our plans of future research.

Context. Computational geometry covers a wide range of fields including geometric modelling using curves and surfaces, computer proofs of geometric theorems, and geometric design software. Recent research in computational geometry has become increasingly sophisticated and tend to abstract from implementation concerns. In particular, the study of the combinatorial properties of geometric problems has resulted in many interesting mathematical results (see, e.g., [1]). However, combinatorial properties are important also for geometric algorithms and their implementations. In fact, such algorithms take logical decisions both on (a) geometric data, such as the coordinates of points, and (b) combinatorial data, such as graphs. The data of type (b) can be represented exactly while those of type (a) have usually a physical origin with some degree of uncertainty. For example, it could be difficult (if not impossible) to decide whether a point lies on the right or on the left of a line identified by other two points. To avoid this problem, the design of geometric algorithms is done by assuming the Random Access Machine (RAM) [10]. This computational model permits real numbers to be represented exactly and to perform exact arithmetic operations on them. Now, geometric data can be assumed to be exact so that deciding whether a point lies on the left or on the right of a given line is also possible. Furthermore, to avoid "strange" cases (e.g. a point lying exactly on a line), algorithms are usually coded in terms of primitive geometric concepts, whose exact characterization depends on the problem to be solved. For example, algorithms computing the convex hull of a set of points in the plane are usually specified in terms of the predicate $cc(p, q, r)$ which holds only if point r is on the left of the line identified by points p and q (going from p to q). Unfortunately, an algorithm designed for the RAM and coded in terms of geometric primitives cannot be implemented reliably in a straightforward way (see, e.g. [7] for an overview on this problem). In fact, there is more than one way in which the RAM and the geometric primitives can fail to be adequate in actual processors. One of the most important sources of inadequacy is the round-off error of floating-point operations. For example, an algorithm for convex hull can return incorrect outputs if the predicate $cc(p,q,r)$ is evaluated to be true and $cc(q,r,p)$ to be false.[1] In other words, the evaluation of geometric primitives may yield mutually contradictory results.

Robustness. An important line of research in computational geometry consists of studying how to correctly implement algorithms designed for the RAM on actual processors. More precisely, implementations of geometric algorithms can be classified into several classes depending on their capability of ensuring precise results. Let us briefly consider the two most common types: *exact* and *robust* implementations. The former are guaranteed to produce an exact result when given an exact input. Indeed, these implementations use a form of exact arithmetic (e.g. an arbitrary precision arithmetic library). Unfortunately, this can result in a substantial and, sometimes unacceptable, slow down. Robust algorithms relax the requirement to obtain exact results and accept that solution are "close enough" in some sense that depends on the application. By dispensing with exact arithmetic, we can build faster implementations which must be supplied with the ability of producing meaningful results despite the fact that the geometric primitives return incorrect results because of inexact arithmetic. No general technique has yet been devised to build robust algorithms; usually, robust implementations are obtained for a certain algorithm by exploiting sophisticated techniques for error analysis. Furthermore, there is the more fundamental difficulty consisting of the fact that the evaluation of geometric primitives may yield mutually contradictory results.

Parsimonious algorithms. An algorithm (computing a purely combinatorial result) will produce a meaningful result if the evaluations of its primitive predicates are wrong but consistent with each other because there exists an input for which those evaluations are correct. This observation leads Fortune [6] to the definition of a sub-class of robust algorithms called *parsimonious*. Intuitively, an algorithm is parsimonious if it never evaluates a primitive predicate whose outcome has already been determined as the formal consequence of previous evaluations of the same predicate on other points. For example, let a, b, and c three non-collinear points and consider again the relation cc. If we establish that c lies on the left of the line going from a to b by computing that $cc(a,b,c)$ holds, then a parsimonious algorithm will derive that b is on the left of the line going from c to a by deduction without computing $cc(c,b,a)$.

In this abstract, we propose a **declarative** method to build parsimonious algorithms based on a combination of SAT solving and (algebraic) computing. The former allows to deduce (if possible) new facts from a knowledge base storing the results of previous computations of the geometric predicates and the latter extends the knowledge base when a fact cannot be deduced. To make our ideas more concrete, we consider the relation cc as the geometric primitive under con-

[1] In fact, if point r is on the left of the line pq, i.e. $cc(p,q,r)$, then point p must be on the left of line qr, i.e. $cc(q,r,p)$.

$$cc(p,q,r) \Longrightarrow cc(q,r,p) \tag{2}$$
$$cc(p,q,r) \Longrightarrow \neg cc(p,r,q) \tag{3}$$
$$cc(p,q,r) \vee cc(p,r,q) \tag{4}$$
$$cc(t,q,r) \wedge cc(p,t,r) \wedge cc(p,q,t) \Longrightarrow cc(p,q,r) \tag{5}$$
$$cc(t,s,p) \wedge cc(t,s,q) \wedge cc(t,s,r) \wedge cc(t,p,q) \wedge cc(t,q,r) \Longrightarrow cc(p,q,r) \tag{6}$$

Fig. 1. The Axioms of *cc*.

sideration. It is well-known that such a relation can be characterised by the sign of the determinant of a matrix. In [9], Knuth provides also an elegant and abstract characterization of such a relation which permits to derive many interesting properties in a purely axiomatic setting. (However, Knuth's characterization of *cc* is partial since it admits also configurations which are not geometrically realizable.) The SAT solver in our combination relies on Knuth's axiomatisation of *cc* to derive new facts for a finite sets of points. By using Knuth's abstract characterization of convex hulls, we will see how a parsimonious algorithm can be obtained as an instance of the proposed combination. We will also see how the proposed combination is flexible enough to handle the case in which configurations of points are described by means of arithmetic constraints. A possible application of this can be to obtain the termination of model checking algorithms for infinite state system by using convex hulls as approximations of the set of reachable states.

The Counterclockwise Relation. Let p, q, and r three **distinct** points in the plane whose Cartesian coordinates are (x_p, y_p), (x_q, y_q), and (x_r, y_r), respectively. We define the *counterclockwise* relation as (twice) the signed area of the triangle identified by p, q, and r, i.e.

$$cc(p,q,r) \Longleftrightarrow \det \begin{pmatrix} x_p & y_p & 1 \\ x_q & y_q & 1 \\ x_r & y_r & 1 \end{pmatrix} = (x_p - x_r)(y_q - y_r) - (x_q - x_r)(y_p - y_r) > 0. \tag{1}$$

We abbreviate the determinant in (1) with $\det(p,q,r)$, which is positive if the points p, q, and r occur in counterclockwise order, negative if they occur in clockwise order, and zero if they are collinear. Another possible interpretation of the relation $cc(p,q,r)$ is that point r lies to the left of the directed line pq identified by going from p to q.

In [9], the *cc* relation is abstractly characterised by the set of five axioms in Figure 1. Axioms (2), (3), and (4) holds for all **distinct** points p, q, and r; axiom (5) for all **distinct** points p, q, r, and t; and axiom (6) for all **distinct** points p, q, r, s, and t. Axiom (2) says that if p, q, and r occur in counter-clockwise order, then all their cyclic permutations do. Axiom (3) requires the counterclockwise relation to be antisymmetric, or if point r lies to the left of the line pq, then it is not possible that point q lies to the left of the line pr. Axiom (4) requires that no three points are collinear (this can be easily checked by using definition (1)).

Axioms (3) and (4) are logically equivalent to $cc(p,q,r) \oplus cc(p,r,q)$ for all distinct points p, q, and r (where $\oplus$ is the symbol for the exclusive-or). (To see this, it is sufficient to derive the conjunctive normal form of the definition of $\oplus$ in terms of $\wedge$ and $\neg$.) Axiom (5) can be rewritten as follows by repeated application of (2) to the first two conjuncts of the antecedent: $cc(q,r,t) \wedge cc(r,p,t) \wedge cc(p,q,t) \Longrightarrow cc(p,q,r)$. Hence, we can read it as if point t lies to the left of the directed lines qr, rp, and pq , then t must be inside the triangle identified by p, q, and r occurring in counterclockwise order. (This can also be checked by expanding definition (1) and routine algebraic manipulations.) Axiom (6), intuitively, says that if points p, q, and r lie to the left of the oriented line ts (first three conjuncts of the antecedent of the implication) and q is to the left of tp and r is to the left of tq (last two conjuncts in the antecedent), then r is to the left of pq. In the following, let $\mathcal{H}$ be the theory axiomatised by axioms (2)–(6).

Any ternary relation satisfying the theory $\mathcal{H}$ is called a *CC system*. It is important to notice that CC systems **do not** capture all the properties of counterclockwise relations between points in the plane. An intuitive argument for this fact is that axioms (2)–(6) involve configurations of at most five distinct points whereas Pappus' theorem states a property about nine points.[2] As a consequence of this, there will be CC systems which can and others cannot arise from actual points in the plane. If it can arise, we call it **realizable**.

In general, the advantage of having an axiomatisation is the possibility to design algorithms based on abstract properties which are easier to reason about and prove correct. For example, it is possible to give the following abstract characterization of the convex hull of a CC system (see page 45 of [9]). The *convex hull* of a CC system is the set of ordered pairs (t,s) of distinct points s.t. $cc(t,s,p)$ holds for all $p \notin \{s,t\}$. Of course, we can define a binary predicate ch s.t. $ch(t,s)$ holds iff the ordered pair (t,s) is in the convex hull by means of the following formulae:

$$ch(t,s) \Longrightarrow \neg ch(s,t) \tag{7}$$

$$ch(t,s) \Longleftrightarrow \forall p.((p \neq s \wedge p \neq t) \Longrightarrow cc(t,s,p)) \tag{8}$$

for each pair of **distinct** points t and s in $\{p_1, ..., p_n\}$. We denote the theory $\mathcal{H} \cup \{(7),(8)\}$ with $\mathcal{H}'$.

By exploiting this definition of convex hull, it is then possible to build algorithms which actually compute the convex hull of a finite set of points satisfying the axioms of CC systems (see [9] for details). Indeed, such algorithms are more general than algorithms that compute convex hulls only with coordinates of points. As a consequence, we should find criteria to check whether a CC system is realizable or not.

Building Parsimonious Algorithms. In principle, it is possible to make an algorithm parsimonious if the primitive geometric predicates can be expressed by polynomial inequalities. In this way, deducing whether the value of a predicate

[2] Pappus' theorem roughly states that if eight triples of points are collinear, then also the ninth triple is collinear.

is the logical consequence of previously established ones can be expressed as a formula of the existential theory of reals [6].[3] This is the case, for example, of the counter-clockwise relation *cc*. (To see this, it is sufficient to recall definition (1).) It can be shown that the problem of checking whether a finite set of points be a realizable CC-system is NP-complete by reducing the problem of determining its realisability to checking the satisfiability of a formula of the theory of reals (see the proof of the corollary at page 22 of [9] for details). So, parsimonious algorithms using the five axioms of CC-systems to deduce facts about the *cc* relation seem to require the solution of NP-hard problems. However, recent research in propositional satisfiability checking (the typical NP-complete problem) shows that it is possible to implement algorithms which are efficient in many practical situations.

Spectacular advances in SAT solvers permit to efficiently solve the satisfiability problem of huge formulae (see [12] for an overview). There are two main streams of research to leverage the advances of SAT solvers. First, reductions to the SAT problem have been devised for a variety of domains ranging from planning (see e.g. [8]) to protocol verification (see e.g. [2]). Second, combinations of SAT solvers with decision procedures for more expressive theories have also been designed and successfully applied (see e.g. [3, 11]). Our methodology to build parsimonious algorithms is a mixture of the two approaches. In fact, we use a combination of SAT solving and algebraic computing to find the satisfying assignments of a ground first-order formula which is obtained by instantiating the axioms of CC-systems and formulae encoding the problem to be solved (e.g. finding a convex hull of a finite set of points in the plane).

Given the set of points of which we want to find the convex hull, we suitably instantiate the axioms and the definition of $\mathcal{H}'$ and we feed the resulting formula φ to a SAT solver. This last is used to enumerate the propositional assignments satisfying φ. Each such assignment is (incrementally) sent to a module which is encharged to evaluate *cc*. If the evaluation of a literal $cc(p, q, r)$ can be made to a sufficient degree of precision by the available implementation of arithmetic operations, then the propositional assignment is accepted or rejected depending whether the truth values of *cc* given by the SAT solver and the arithmetic module are the same or not. If the two modules return the same result then the truth value of the actual literal is learnt by the SAT solver and the next literal in the assignment is considered. Otherwise, the literal is marked as unknown and its truth value will be determined by the SAT solver by exploiting the trustable results of the arithmetic module. If too many values are marked as unknown so that there is no way to derive the truth value of any of them by the SAT solver (i.e. by purely deductive means), then one of the unknown literals is evaluated by a module implementing exact arithmetic. All these operations are repeated until the two modules agree on a unique propositional assignment. By the definition of the predicate *ch*, it is then easy to extract the convex hull of the input points. If we replace the module implementing arithmetic operations with constraint solvers for classes of arithmetic constraints (e.g. the Fourier-Motzkin algorithm

[3] The decision problem for this theory is in the class PSPACE [5].

for linear constraints), then we easily lift the algorithm to the case in which the points are identified by constraints.

Future Work. The work described above is ongoing and we plan to develop it in several directions. The most important concerns the implementation. We plan to implement the proposed algorithm by adapting the code of a state-of-art SAT solvers. We intend to study the performances of the system by using phase transition techniques (see e.g. [3]). We will also compare our system with state-of-the-art implementations of ad-hoc algorithms to compute the convex hull in order to have a clear picture about the scalability of the proposed approach. Finally, we want to assess the impact on the performances of instances of axioms (3) and (4) which generate the CNF of an exclusive-or which is known to be problematic for SAT solvers (see, e.g. [4] for a discussion of this issue).

Finally, we will also look at how our approach can be adapted to compute convex hulls in three dimensions, Delaunay triangulations, and their generalisation to higher dimension along the lines sketched in [9].

References

1. Special Issue on the Complexity of Arrangements. Journal of Discrete and Computational Geometry, Volume 5, 1990.
2. A. Armando and L. Compagna. Automatic SAT-compilation of Protocol Insecurity Problems via Reduction to Planning. In *Proceedings of the Joint International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2002)*, 2002.
3. Alessandro Armando, Claudio Castellini, and Enrico Giunchiglia. SAT-based procedures for temporal reasoning. In *ECP*, pages 97–108, 1999.
4. Peter Baumgartner and Fabio Massacci. The Taming of the (X)OR. In *Computational Logic – CL 2000*, volume 1861, pages 508–522. Springer, 2000.
5. J. Canny. Some Algebraic and Geometric Computations in PSPACE. In *Proc. of 20th Annual Symp. on the Theory of Computing*, pages 460–467. ACM, 1988.
6. S. Fortune. Stable Maintenance of Point Set Triangulations in Two Dimensions. In *Proc. of 30th Annual Symposium on Foundations of Computer Science*, pages 494–499. IEEE Computer Society Press, 1989.
7. C. M. Hoffmann. The Problems of Accuracy and Robustness in Geometric Computation. *The Computer Journal*, pages 31–41, March 1989.
8. Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In Howard Shrobe and Ted Senator, editors, *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 1194–1201, Menlo Park, California, 1996. AAAI Press.
9. D. E. Knuth. *Axioms and Hulls*, volume 606 of *LNCS*. Springer-Verlag, 1992.
10. F. P. Preparata and M. I. Shamos. *Computational Geometry*. Sprigner-Verlag, 1985.
11. R. Sebastiani. Integrating sat solvers with math reasoners: Foundations and basic algorithms, 2001.
12. L. Zhang and S. Malik. The quest for efficient Boolean satisfiability solvers. In *Proc. Conference on Automated Deduction (CADE02)*, volume 2392 of *LNCS*, pages 295–313. Springer-Verlag, 2002.

Rings and Modules in Isabelle/HOL

Hidetsune KOBAYASHI[1], Hideo SUZUKI[2], and HirokazuMURAO[3]

[1] Department of Mathematics, Nihon University
[2] Tokyo Institute, Polytechnic University
[3] Department of Computer Science, University of Electro-Communications

1 Introduction

We report formalization of rings and modules over rings in Isabelle/HOL. Topics in this report are Chinese remainder theorem of general type, local rings, Jacobson radicals, finitely generated modules and Nakayama lemma.

2 Formalization of groups

To show how to formalize mathematical objects in Isabelle/HOL, we present formalization of a group:

```
record ' a grouptype =
pcarrier :: "'a set "
bOp1 :: "['a, 'a]⇒ 'a"
iOp1 :: "'a⇒ 'a "
unit1 :: "'a"

constdefs
Group :: "('a, 'more) grouptype_scheme ⇒ bool "
"Group G == (bOp1 G) ∈ pcarrier G → pcarrier G → pcarrier
G
∧ (iOp1 G) ∈ pcarrier G → pcarrier G ∧
(unit1 G) ∈ pcarrier ∧
∀ x ∈ pcarrier G. ∀ y ∈ pcarrier G. ∀ z ∈ pcarrier G.
(bOp1 G (unit1 G) x = x) ∧
(bOp1 G (iOp1 G x) x = unit1 G) ∧
(bOp1 G (bOp1 G x y) z = bOp1 G (x) (bOp1 G y z)))"
```

See a source code [2] written by Florian Kammueller and L. C. Paulson (old version). "bOp1 G x y" is the binary operator of the group, and this is expressed as $x \cdot_G y$ if we have two more lines

```
syntax "BOP1" :: "['a, ('a, 'more) grouptype_scheme, 'a] ⇒
'a" ("(3 _ ·_ _)" [80,80,81]80)
translations "x ·G y" == "bOp1 G x y"
```

Here, "record" defines the type of a group and constdefs gives a definition of the group. "pcarrier" is the underlying set of the group, bOp1 is the inverse operator, and unit1 is the unit of the group.

Subgroup of G is defined as a subset of "pcarrier G" having bOp1 closedness and iOp1 closedness, and so subgroup is not a group in Isabelle/HOL. We introduce "Grp" to make a group with subgroup H as "pcarrier":

```
constdefs
Grp :: "[('a, 'more) grouptype_scheme, 'a set ] ⇒ 'a
grouptype" "Grp G H == (|pcarrier = H, bOp1 = bOp1 G, iOp1
= iOp1 G, unit1 = unit1 G|)"
```

Using this Grp, Zassenhaus theorem is formalized as

```
theorem Zassenhaus: "[| Group G; H ≪ G; H1 ≪ G; K ≪ G;
K1 ≪ G; H1 ◁ Grp G H; K1 ◁ Grp G K |] ⟹
((Grp G (H1 ⋄G (H ∩ K)))/ (H1 ⋄G (H ∩ K1))) ≅
         ((Grp G (K1 ⋄G (K ∩ H)))/(K1 ⋄G (K ∩ H1)))"
```

Here, H ≪ G means H is a subgroup of G, K1 ◁ Grp G K means K1 is a normal subgroup of Grp G K and `K1 ⋄G (K ∩ H)` is the set $\{kh | k \in K1, h \in K \cap H\}$. In some textbooks, this set is written simply as $\mathrm{H}_1(\mathrm{H} \cap \mathrm{K})$ and the theorem is written as

Theorem. Let G be a group. Let H, H_1, K and K_1 be subgroups of G such that $\mathrm{H}_1 \lhd \mathrm{H}$ and $\mathrm{K}_1 \lhd K$. Then we have an isomorphism

$$\mathrm{H}_1(\mathrm{H} \cap \mathrm{K})/\mathrm{H}_1(\mathrm{H} \cap \mathrm{K}_1) \cong \mathrm{K}_1(\mathrm{K} \cap \mathrm{H})/\mathrm{K}_1(\mathrm{K} \cap \mathrm{H}_1)$$

To treat rings and modules, we prepare "agrouptype" for abelian groups:

```
record 'a agrouptype =
carrier :: "'a set"
abOp1 :: "['a, 'a ] ⇒ 'a"
aiOp1 :: "'a ⇒ 'a"
aunit1 :: "'a"
```

```
constdefs
AGroup :: "('a, 'more) agrouptype_scheme ⇒ bool"
"AGroup G == (abOp1 G)∈ carrier G → carrier G → carrier G
∧
(aiOp1 G) ∈ carrier G → carrier G ∧
(aunit1 G) ∈ carrier G ∧
(∀x ∈ carrier G. ∀y ∈ carrier G. ∀ z ∈ carrier G.
(abOp1 G (aunit1 G) x = x) ∧ (abOp1 G (aiOp1 G x) x =
aunit1
G) ∧ (abOp1 G (abOp1 G x y) z = abOp1 G (x) (abOp1 G y z))
∧
(abOp1 G x y = abOp1 G y x))"
```

The main reason to introduce "agrouptype" is that we will need both multiplicative groups and abelian groups later.

3 Rings, ideals and Chinese remainder theorem

A ring is an abelian group with one more binary operation "multiplication". "ringtype" is given as

```
record 'a ringtype = "'a agrouptype" +
bOp2 :: "['a, 'a ] ⇒ 'a"
unit2 :: "'a"
```

A formalized definition of a ring is given similarly to a group definition.

Zorn lemma is formalized by Jacques D. Fleuriot, and his formalization enables us to treat maximal ideals. The well known property "the nilradical is equal to the intersection of all prime ideals" is formalized by using Fleuriot's formalization of Zorn lemma.

We present two lemmas concerning maximal ideals.

```
lemma local_ring_diff:"[| Ring R; not ZeroRing R; ideal R
mx; mx ≠ carrier R;
∀a∈ (carrier R - mx). unit R a |] ⟹ local_ring R ∧
maximal_ideal R mx"
```

This lemma states "Let R be a non-zero ring. Let mx be a proper ideal of R. If any element not contained in mx is a unit, then R is a local ring and mx is the maximal ideal of R".

Jacobson radical is the intersection of all maximal ideals. Following lemma is used to prove Nakayama's lemma.

```
lemma J_rad_unit:"[|Ring R; not ZeroRing R; x ∈ J_rad R|]
⟹ ∀y. (y∈ carrier R ⟶ unit R (1R +R (-Rx) •R y))"
```

This lemma states " Let R be a non-zero ring. If x is included in the Jacobson radical, then for any y in R $1 - x\ y$ is a unit".
To formalize Chinese remainder theorem, we formalize direct products.

```
constdefs

prodAG::"['i set, 'i ⇒ ('a, 'more) agrouptype_scheme] ⇒
('i ⇒ 'a) agrouptype"

"prodAG I A == (| carrier = carr_prodAG I A,
abOp1 = prod_bOp1 I A, aiOp1 = prod_iOp1 I A,
aunit1 = prod_unit1 I A|)
```

In this definition, I is an index set and "carr_prodAG I A" is the carrier of the direct product of abelian groups. Direct product of rings is defined similarly.

Chinese remainder theorem of general type is formalized as

```
theorem Chinese_remThm:"[| Ring R; (∀k∈Nset (Suc n). ideal
R (J k));
∀k∈Nset (Suc n). B k = QRing R (J k);
∀k∈Nset (Suc n). S k = pj R (J k);
∀i∈Nset (Suc n). ∀j∈Nset (Suc n). (i ≠j ⟶ coprime_ideals
R (J i) (J j)) |] ⟹
R /r(⋂ {J k | k. k∈Nset (Suc n)} ≅  r∏Nset (Suc n) B"
```

Here, Nset (Suc n) is a set of integers from 0 up to $n+1$, and QRing is the residue class ring.

4 Modules over a ring R

A module is an abelian group having scalar multiplication with a ring.

Direct product of modules is formalized as direct product of rings, and a formalization of direct sum is derived from it by adding a simple condition.

Let H be a subset of a module M over R. We call linear span, the smallest submodule containing H. It is equal to a set of all linear combinations of elements of H. The Linear span is formalized as:

```
constdefs
linear_combination::"[('r, 'm) ringtype_scheme, ('a, 'r,
'm1) moduletype_scheme, nat] ⇒ (nat ⇒ 'r) ⇒ (nat ⇒ 'a)
⇒ 'a"
"linear_combination R M n s m == eSum M (λj. (s j) ⋆M (m
j)) n"

linear_span::"[('r, 'm) ringtype_scheme, ('a, 'r, 'm1)
moduletype_ scheme, 'r set, 'a set] ⇒ 'a set"
"linear_span R M A H == if H = {} then {0M} else {x. ∃n. ∃f
∈ Nset n → H. ∃s∈Nset n → A. x = linear_combination R M n
s f}"
```

Note that "linear_span R M A H" is a submodule generated by H with coefficients in an ideal A of the ring R. If we take carrier R as A, then the linear_span coincides the ordinary linear span.

A finitely generated submodule is a module having a finite set H such that " Linear_span R M (carrier R) H = carrier M". Nakayama lemma is

> Let M be a finitely generated module over a ring R, and let A be an ideal contained in the Jacobson radical. If A M = M then M = 0.

This is formalized as

```
lemma NAK:"[| Ring R; R Module M; M fgover R; ideal R A; A
⊆ J_rad R; A ⊙R M = carrier M |] ⟹ carrier M = {0M}"
```

Here, A $\odot_{\mathrm{R}}$ M is `linear_span R M A (carrier M)`, and `M fgover R` means M is finitely generated over R.

There are two ways to prove this lemma, one is using determinant trick and another is using a generator with the least number of elements (see [1]). We formalized the letter proof.

To formalize Nakayama lemma, we have to sum up coefficients of similar terms. More precisely, we need following formalization:

```
lemma finite_lin_span:"[|Ring R; R Module M; ideal R A; h
∈ Nset n → carrier M; s ∈ Nset na → A; f∈Nset na → h `
Nset n |] ⟹ ∃t∈Nset n → A.
linear_combination R M na s f = linear_combination R M n t
h"
```

Corollary to Nakayama lemma is formalized as

```
lemma NAK1:"[| Ring R; ¬ ZeroRing R; R Module M; M fgover
R; Submodule R M N; ideal R A; A ⊆ J_rad R; carrier M = A
⊙R M +M N |] ⟹ carrier M = N"
```

Finally we give a short note on type restriction of Isabelle/HOL. Points are (a) type restriction is too strong and mathematical propositions become a little complicated in formalized expressions, (b) because of type restriction, we cannot express n objects having n independent types.

(a) For example, even two modules have different type, say ('a, 'r) moduletype and ('b, 'r) moduletype respectively, these two may happen to be isomorphic. Ring A and A/0 is isomorphic, but the former has type "'r ringtype" and that of the latter one is "'r set ringtype".

Because of type restriction, we cannot take a ring R as an R-module. So we introduced a record bModule

```
bModule ::"[('r1, 'm1) ringtype_scheme, ('r2, 'm2)
ringtype_scheme, ('a, 'r1, 'r2, 'more) bmoduletype_scheme]
⇒ bool"
```

This enables us to treat free modules over R.

(b) We cannot formalize exact sequence of any length.

References

[1] M. F. Atiyah and I. G. Mcdonald, Introduction to commutative algebra. Addison-Wesley, 1969.

[2] Isabelle/src/HOL/GroupTheory/Group.thy

[3] Isabelle/src/HOL/GroupTheory/Ring.thy

[4] L. Chen, H. Kobayashi, H. Murao, H. Suzuki. A Machine Proof of the Proposition "Ideal $\subseteq \cup_i$ PrimeIdeal$_i \Longrightarrow$ Ideal $\subseteq$ PrimeIdeal$_i$", RIMS Computer Algebra - Algorithms, Implementations and Applications Vol. 1295, pages 42-50, 2003.

[5] L. Chen, H. Kobayashi, H. Murao, H. Suzuki. Notes on formalizing induction

on the number of sets. In S. Colton and V. Sorge, editors, Second Workshop on the Role of Automated Deduction in Mathematics: RADM. In conjunction with CADE-18, pages 11-23, 2002.

Editor's note
This paper could not be edited to comply with the proceedings style

Exploring an Algorithm for Polynomial Interpolation in the *Theorema* System

*Wolfgang Windsteiger**

RISC Institute, University of Linz
A-4232 Hagenberg, Austria

Abstract

We present a case study using the *Theorema* system to explore an algorithm for polynomial interpolation. The emphasis of the case study lies on formulating mathematical knowledge in *one* language that appears in its syntax close to common mathematical language but is precise enough to formulate all details necessary for proving. Moreover, the language allows the computation of concrete examples without any further translation into an executable language.

---

* This work has been supported by the "Spezialforschungsbereich for Numerical and Symbolic Scientific Computing" (SFB F013) at the University of Linz and the european union "CALCULEMUS Project" (HPRN-CT-2000-00102). The author would like to thank B.Buchberger and M.Rosenkranz for many valuable discussions during the case study that led to this paper.

---

1 Introduction

Existing mathematical software systems (e.g. Mathematica, MAPLE, Gap etc.) have made big progress over the past decades by providing the users with comprehensive libraries of sophisticated algorithms in various areas of mathematics. In parallel, basically independently, the past decades have also produced enormous progress in the automation of the proving activity of mathematicians. These approaches, however, have put their emphasis mainly on proving isolated theorems, where systems like Otter, Spass, or Vampire are rather successful.

The challenge for the future is the theoretical foundation, the design, and implementation of *integral software systems* that guide, support, and at least partially automate the entire process of inventing, proving, and applying mathematical knowledge using mathematical knowledge and method libraries and, as a result, expanding these libraries by the result of this process. Recently, this integral view and research program for the next generation of mathematical software systems has been named “Mathematical Knowledge Management” (MKM) in the first international workshop on MKM, Sept. 14-16, 2001, organized at RISC-Linz by B. Buchberger, see [MKM 03].

From the very beginning the *Theorema* project was meant to give a logical and software technological frame for the entire mathematical knowledge management as an integral coherent process with the following key design objectives and ideas:

- The three main activities of mathematics—proving, computing, and solving—should all be available in one logical and software technological frame. Moreover, the natural interplay between proving, computing, and solving should be supported by the system.

- Domains, functors and categories as a natural and powerful structuring mechanism for generic build-up of systematic knowledge and methods.
- Preference to special proving, simplifying, and solving algorithms for special mathematical theories as opposed to a one-method-approach to proving all of mathematics (e.g. resolution method) with the possibility to link powerful and provenly correct algebraic algorithms (e.g. Cylindrical Algebraic Decomposition for quantifier elimination, Gröbner bases method for systems of algebraic equations, Risch's algorithm for symbolic integration, or Zeilberger's algorithm for sum identities).
- High usability and attractiveness for the working mathematician by using various software technological advances, e.g. flexible syntax imitating the usual textbook-style, proof presentation with high readability and postprocessing capabilities.
- Knowledge management tools for the construction, maintenance, and modification of large mathematical knowledge bases together with system support for integral theory exploration: failure analysis for proofs and conjecture generation based on failure analysis; build-up of libraries of problem types, knowledge types and algorithm types and their systematic use in the theory exploration process.

In this paper, we would like to demonstrate part of the currently available features in *Theorema* in a case study, namely polynomial interpolation. In this case study, we show: 1) How the domain of univariate polynomials can be built up in generic form using the functor construct available in *Theorema*. 2) We demonstrate how the problem of interpolation can be specified in this setting. 3) How a special solution of the interpolation problem, namely by the Neville algorithm, can be formulated. 4) How its correctness proof can be given. 5) How the algorithm can finally be applied to concrete problems.

With this case study we want to demonstrate the following aspects, which play a crucial role in an integral view of the mathematical knowledge management process: 1) Problem specification, algorithmic formulation, correctness proof, and computation (application to concrete examples) can be done within the one and uniform logic and system frame of *Theorema*. 2) The possibility for formulating mathematical knowledge and methods in a generic way that guarantees applicability and re-usability in a wide range of (hierarchically built-up) domains. 3) Attractive choice of syntax, which is close to the common usage of notation in mathematical textbooks and at the same time is formally rigorous in the sense that all formulae (including the algorithms) are just formulae in the underlying predicate logic. We want to particularly emphasize the didactic challenge in this context: on the one hand, every detail must be spelled out unambiguously whereas, on the other hand, we want to stay close to common use (and often ab-use) of mathematical notation used in mathematical texts.

In this paper, we do not yet talk about possibilities in *Theorema* for guiding and supporting the invention process. However, note for example, that proofs for elementary properties of e.g. polynomial evaluation, which are needed for polynomial interpolation, are naturally suggested by the structure of the polynomial domain as defined in the polynomial functor. However, we would also like to emphasize that systematic methods for mathematical exploration, in our view, are not only a desirable goal for completely automating the invention process (which will never be possible by the inherent incompleteness of mathematics) but are a very reasonable and worthwhile research goal for improving the didactics and heuristics of mathematics. In concrete terms this means that at certain stages in the invention process instead of getting support from the system the user may also interact with the system by allowing the user to guide the prover or suggesting the prover the general structure of an algorithm.

This case study is taken from lecture notes used in courses, whose goal is to present the entire content of the first year of mathematics study in an algorithmic fashion. The *Theorema* language turns out to provide a suitable frame for these courses, because the entire mathematical knowledge including all algorithms can be formulated in a style, which later allows *proving* all the properties in subsequent courses. Similar case studies have been initiated for other topics such as Gaussian elimination or Gröbner bases theory.

2 The Polynomial Functor

We present a case study in the domain of univariate polynomials over a field K. Polynomials can be defined to be infinite K-sequences with only finitely many non-zero elements, i.e. the (infinite) direct sum of (infinitely many copies of) the coefficient field. Thus, for each such sequence there must be an index, such that the sequence consists of only zeroes after this index. This is an appropriate setting for a computer-representation of polynomials, since it allows to naturally represent a polynomial by a K-tuple of its coefficients up to the last non-zero entry in the sequence.

In the *Theorema* language, the domain of univariate polynomials over a coefficient field K can be introduced nicely by a *Functor*. Functors are a well-known concept (e.g. in category theory) and the hierachical construction of mathematical domains by functors has already been used in Computer Algebra systems (the use of domains and categories is one of the distinctive design features of the well-known AXIOM system, see [AXIOM]). The algorithmic nature of functors as introduced in the *Theorema* system (see [Buchberger 96a], [Buchberger 96b], and [Windsteiger 99]) relates to how functors are available in the programming language ML. In general, a functor allows to construct a new domain from an already existing domain. In the concrete case, we assume a domain K and construct the domain of polynomials over K, named Poly$[K]$, by defining the characteristic property for the elements in Poly$[K]$ and by defining operations in Poly$[K]$ based on available operations in the underlying domain K. Note that we will present here only part of the functor, namely just those definitions that are relevant for further discussion on the polynomial interpolation algorithm presented in Section 4. The functor definition shown in Figure 1 must be read as follows: The domain Poly$[K]$ is such a domain P, where, for any p, q, n, a, the following operations are defined:

- $\underset{P}{\in}[p]$ (p is an element in P) iff p is a tuple of positive length with elements from K.
- $\underset{P}{x}$ (a new constant x in P) is the tuple $\left\langle \underset{K}{0}, \underset{K}{1} \right\rangle$.
- $\underset{P}{\deg}[p]$ (the degree of p in P) is either 0 or it is such an i between 1 and $|p|$ such that … . (The "such a-quantifier" $\underset{i=1,\ldots,|p|}{\ni} \ldots$ is a special language construct available in the *Theorema* language, which stands for "such an i between 1 and $|p|$ satisfying the property …". It provides a formal frame for giving *implicit function definitions*.)
- etc.

The constant x in the polynomial domain plays exactly the role of the "polynomial indeterminate" x when thinking of polynomials as "arithmetic terms" of the form $\sum_{k=0}^{n} p_i\, x^i$. In the functor notation all function, predicate, and object constants carry the domain, for which they are defined, as an underscript, e.g. $\underset{\mathrm{Poly}[K]}{-}$ for subtraction in the domain of polynomials as opposed to $\underset{K}{-}$ for subtraction in the domain K. In the remainder of this paper, all text in gray boxes is *Theorema* input or output as it appears in a *Theorema* session. The syntax used—including all special symbols and typesetting facilities—is machine-readable and the *Theorema* parser translates it unambiguously into *Theorema*'s internal representation.

Definition["Polynomial Domain", any[K],

Poly[**K**] := Functor[P, any[p, q, n, a],

$$\underset{P}{\in}[p] \Leftrightarrow \left(\text{is-tuple}[p] \bigwedge |p| > 0 \bigwedge \underset{i=1,\ldots,|p|}{\forall} \underset{K}{\in}[p_i]\right)$$

$$\underset{P}{x} := \left\langle \underset{K}{0}, \underset{K}{1} \right\rangle$$

$$\underset{P}{\deg}[p] := \begin{cases} 0 & \Leftarrow \underset{j=1,\ldots,|p|}{\forall} \left(p_j = \underset{K}{0}\right) \\ \underset{i=1,\ldots,|p|}{\ni} \left(\left(p_i \neq \underset{K}{0}\right) \bigwedge \underset{j=i+1,\ldots,|p|}{\forall} \left(p_j = \underset{K}{0}\right)\right) - 1 & \Leftarrow \text{otherwise} \end{cases}$$

$$\underset{P}{\text{coef}}[p, n] := \begin{cases} p_{n+1} & \Leftarrow n \geq 0 \bigwedge n \leq \underset{P}{\deg}[p] \\ \underset{K}{0} & \Leftarrow \text{otherwise} \end{cases}$$

$$\underset{P}{\text{const}}[a] := \langle a \rangle$$

$$\underset{P}{\text{canonic}}[p] := \left\langle p_i \;\Big|_{i=1,\ldots,\underset{P}{\deg}[p]+1} \right\rangle \qquad]]$$

$$p \underset{P}{-} q := \underset{P}{\text{canonic}}\left[\left\langle \underset{P}{\text{coef}}[p, i] \underset{K}{-} \underset{P}{\text{coef}}[q, i] \;\Big|_{i=0,\ldots,\text{Maximum}\left[\underset{P}{\deg}[p], \underset{P}{\deg}[q]\right]} \right\rangle\right]$$

$$p \underset{P}{*} q := \left\langle \underset{j=0,\ldots,i}{\textstyle\sum_K} \underset{P}{\text{coef}}[p, j] \underset{K}{*} \underset{P}{\text{coef}}[q, i-j] \;\Big|_{i=0,\ldots,\underset{P}{\deg}[p]+\underset{P}{\deg}[q]} \right\rangle$$

$$p \underset{P}{/} a := \left\langle \underset{P}{\text{coef}}[p, i] \underset{K}{/} a \;\Big|_{i=0,\ldots,\underset{P}{\deg}[p]} \right\rangle$$

$$\underset{P}{\text{eval}}[p, a] := \underset{i=0,\ldots,\underset{P}{\deg}[p]}{\textstyle\sum_K} \underset{P}{\text{coef}}[p, i] \underset{K}{*} a^i$$

Figure 1: The functor defining the domain of univariate polynomials.

3 Problem Specification: Polynomial Interpolation

Given a polynomial p over K and two tuples x and a, one might ask, whether p evaluates (in Poly[K]) to a_i at x_i (for all $i = 1, \ldots, |x|$), i.e. whether p is an interpolating polynomial for x and a in Poly[K]. This consideration is natural because then the "polynomial function associated with p" would run through all the given points $\langle x_i, a_i \rangle$, which is a crucial property for many applications in mathematics (e.g. several methods for solving equations are based on iteratively solving equations for interpolating polynomials). In *Theorema*, this property can be expressed as follows:

Definition["Interpolating polynomial: characterization", any[p, x, a, K],

$$\text{IsInterpolatingPolynomial}[p, x, a, K] :\Leftrightarrow \left(\underset{\text{Poly}[K]}{\in}[p] \bigwedge \underset{\text{Poly}[K]}{\deg}[p] \leq |x| - 1 \bigwedge \underset{i=1,\ldots,|x|}{\forall} \left(\underset{\text{Poly}[K]}{\text{eval}}[p, x_i] = a_i\right)\right)]$$

Under certain restrictions—the tuples x and a must be non-empty and have equal length and the elements of x must be mutually distinct—it can be shown that for given x, a, and K there exists a unique polynomial p over K of degree less equal $|x| - 1$ such that IsInterpolatingPolynomial[p, x, a, K]. Of course, it is then desirable to come up with an *algorithm* that computes the interpolating polynomial for given tuples x, a and coefficient field K.

4 Solution to the Interpolation Problem: Neville Algorithm

An ad-hoc solution for an interpolation algorithm can immediately be extracted from the proof of unique existence of the interpolating polynomial. The proof of this fact can be reduced to prove solvability of a system of linear equations, which is always guaranteed under the given restrictions on x and a. The interpolating polynomial can then be computed by solving the linear system. However, there are better algorithms for finding the interpolating polynomial, for instance the *Neville algorithm*, which proceeds by *recursion* over the tuples x and a. Written in *Theorema* the algorithm is given as follows:

Algorithm["Neville", any[x, a, x0, $\bar{x}$, xn, a0, $\bar{a}$, an, K],

$$\text{NevillePolynomial}[\langle x\rangle, \langle a\rangle, K] = \underset{\text{Poly}[K]}{\text{const}}[a]$$

$$\text{NevillePolynomial}[\langle x0, \bar{x}, xn\rangle, \langle a0, \bar{a}, an\rangle, K] =$$

$$\left(\left(\left(\underset{\text{Poly}[K]}{x} \underset{\text{Poly}[K]}{-} \underset{\text{Poly}[K]}{\text{const}}[x0]\right) \underset{\text{Poly}[K]}{*} \text{NevillePolynomial}[\langle \bar{x}, xn\rangle, \langle \bar{a}, an\rangle, K] \underset{\text{Poly}[K]}{-} \left(\underset{\text{Poly}[K]}{x} \underset{\text{Poly}[K]}{-} \underset{\text{Poly}[K]}{\text{const}}[xn]\right) \underset{\text{Poly}[K]}{*} \text{NevillePolynomial}[\langle x0, \bar{x}\rangle, \langle a0, \bar{a}\rangle, K]\right) \underset{\text{Poly}[K]}{/} \left(xn \underset{K}{-} x0\right)\right]$$

5 Correctness of the Algorithm

The correctness theorem for the Neville algorithm written in *Theorema* syntax is as follows:

Theorem["Neville polynomial is interpolating polynomial", any[is-tuple[x], a, K], with[$|x| > 0 \wedge |a| = |x|$]
IsInterpolatingPolynomial[NevillePolynomial[x, a, K], x, a, K]]

For automatically proving a formula *for all tuples* x, we can use the *tuple induction prover* available in the *Theorema* system. This prover implements a special prove technique available for tuples, namely Noetherian induction. In order to call this prover, we issue the *Theorema* command

Prove[Theorem["Neville polynomial is interpolating polynomial"], using → KB, by → TupleInduction] ,

where KB contains the knowledge base of auxiliary assumptions needed for the proof. We will refer to required knowledge from KB in the proof later. The tuple induction prover comes up with a successful and complete proof. Due to space limitations, we show only the key steps of this proof (text in boxes contains explanation of the prove techniques applied, *all the rest*—including formula labels, references, and intermediate text—is generated completely automatically by the prover).

Since x is a tuple an induction over x is set up. Since nothing is known about a and K the prover chooses a, K arbitrary but fixed.

Induction base: $x = \langle x1\rangle$ for arbitrary but fixed x1 . We have to show:

(1) $|a| = 1 \Rightarrow$ IsInterpolatingPolynomial[NevillePolynomial[⟨x1⟩, a, K], ⟨x1⟩, a, K] ,

We assume

(2) $|a| = 1$,

and show

(3) IsInterpolatingPolynomial[NevillePolynomial[⟨x1⟩, a, K], ⟨x1⟩, a, K] .

From (2), we can infer:

(4) a = ⟨a1⟩ ,

for some new constant a1 .

Formula (3), using (4) and (Algorithm (Neville)), is implied by:

(5) IsInterpolatingPolynomial[$\underset{\text{Poly}[K]}{\text{const}}$[a1], ⟨x1⟩, ⟨a1⟩, K] ,

which, using (Definition (Polynomial Domain)), is implied by:

(6) $\text{IsInterpolatingPolynomial}[\langle a1\rangle, \langle x1\rangle, \langle a1\rangle, K]$,

which, using (Definition (Interpolating polynomial: characterization)), is implied by:

(7) $$\underset{\text{Poly}[K]}{\in}[\langle a1\rangle] \bigwedge \underset{\text{Poly}[K]}{\text{deg}}[\langle a1\rangle] \le |\langle x1\rangle| - 1 \bigwedge \underset{i=1,\ldots,|\langle x1\rangle|}{\forall}\left(\underset{\text{Poly}[K]}{\text{eval}}[\langle a1\rangle, \langle x1\rangle_i] = \langle a1\rangle_i\right),$$

Formula (7) can now be easily verified by expanding the polynomial operations defined in the functor.

Induction hypothesis: We assume for arbitrary but fixed $n \ge 1$

(8) $\underset{x}{\forall}(|x| = n \wedge |a| = |x|) \Rightarrow \text{IsInterpolatingPolynomial}[\text{NevillePolynomial}[x, a, K], x, a, K]$,

and show

(9) $(|x| = n + 1 \wedge |a| = |x|) \Rightarrow \text{IsInterpolatingPolynomial}[\text{NevillePolynomial}[x, a, K], x, a, K]$.

We assume

(10) $|x| = n + 1$,

(11) $|a| = |x|$,

From (10) and (11), we can infer:

(12) $x = \langle x0, \overline{x}, xn\rangle$,

(13) $a = \langle a0, \overline{a}, an\rangle$,

for new constants x0, xn, a0, an and new constant sequences $\overline{x}$ and $\overline{a}$ of length $n - 1$.

The prover guesses this particular structure for representing x and a from the definition of NevillePolynomial.

It remains to show

(14) $\text{IsInterpolatingPolynomial}[\text{NevillePolynomial}[\langle x0, \overline{x}, xn\rangle, \langle a0, \overline{a}, an\rangle, K], \langle x0, \overline{x}, xn\rangle, \langle a0, \overline{a}, an\rangle, K]$.

Formula (14), using (Algorithm (Neville)), is implied by:

(15) $$\text{IsInterpolatingPolynomial}\Big[\Big(\Big(\underset{\text{Poly}[K]}{x} \underset{\text{Poly}[K]}{-} \underset{\text{Poly}[K]}{\text{const}}[x0]\Big) \underset{\text{Poly}[K]}{*} \text{NevillePolynomial}[\langle \overline{x}, xn\rangle, \langle \overline{a}, an\rangle, K] \underset{\text{Poly}[K]}{-}$$
$$\Big(\underset{\text{Poly}[K]}{x} \underset{\text{Poly}[K]}{-} \underset{\text{Poly}[K]}{\text{const}}[xn]\Big) \underset{\text{Poly}[K]}{*} \text{NevillePolynomial}[\langle x0, \overline{x}\rangle, \langle a0, \overline{a}\rangle, K]\Big) \underset{\text{Poly}[K]}{/} \Big(xn \underset{K}{-} x0\Big),$$
$$\langle x0, \overline{x}, xn\rangle, \langle a0, \overline{a}, an\rangle, K\Big]$$

Membership in the polynomial domain and the degree bound for the interpolating polynomial are not too difficult to prove. For proving the evaluation property we need some auxiliary knowledge about polynomial evaluation, such as e.g. $\text{eval}[p + q, a] = \text{eval}[p, a] + \text{eval}[q, a]$, which can be proven by *another special prover*, which can handle formulae containing the $\sum$-quantifier. In our approach of theory exploration, we suppose that this knowledge has already been proven in a previous exploration phase and is for this proof available in the knowledge base KB. After several simplifications we arrive at the following formula to be proved:

$$\underset{i=1,\ldots,n+1}{\forall}\Bigg(\Big(\underset{\text{Poly}[K]}{\text{eval}}\Big[\Big(\underset{\text{Poly}[K]}{x} \underset{\text{Poly}[K]}{-} \underset{\text{Poly}[K]}{\text{const}}[x0]\Big), \langle x0, \overline{x}, xn\rangle_i\Big] \underset{K}{*} \underset{\text{Poly}[K]}{\text{eval}}\Big[$$
$$\text{NevillePolynomial}[\langle \overline{x}, xn\rangle, \langle \overline{a}, an\rangle, K], \langle x0, \overline{x}, xn\rangle_i\Big] \underset{K}{-} \underset{\text{Poly}[K]}{\text{eval}}\Big[\Big(\underset{\text{Poly}[K]}{x} \underset{\text{Poly}[K]}{-} \underset{\text{Poly}[K]}{\text{const}}[xn]\Big), \langle x0, \overline{x}, xn\rangle_i\Big] \underset{K}{*}$$
$$\underset{\text{Poly}[K]}{\text{eval}}[\text{NevillePolynomial}[\langle x0, \overline{x}\rangle, \langle a0, \overline{a}\rangle, K], \langle x0, \overline{x}, xn\rangle_i]\Big) \underset{K}{/} \Big(xn \underset{K}{-} x0\Big) = \langle a0, \overline{a}, an\rangle_i\Bigg)$$

Since $\langle x0, \overline{x}, xn\rangle_i$ (and $\langle a0, \overline{a}, an\rangle_i$) can be simplified in case $i = 1$ or $i = n + 1$ a case distinction is now made:

Case $i = 1$: We have to show

$$\Big(\underset{\text{Poly}[K]}{\text{eval}}\Big[\Big(\underset{\text{Poly}[K]}{x} \underset{\text{Poly}[K]}{-} \underset{\text{Poly}[K]}{\text{const}}[x0]\Big), x0\Big] \underset{K}{*} \underset{\text{Poly}[K]}{\text{eval}}[\text{NevillePolynomial}[\langle \overline{x}, xn\rangle, \langle \overline{a}, an\rangle, K], x0] \underset{K}{-}$$
$$\underset{\text{Poly}[K]}{\text{eval}}\Big[\Big(\underset{\text{Poly}[K]}{x} \underset{\text{Poly}[K]}{-} \underset{\text{Poly}[K]}{\text{const}}[xn]\Big), x0\Big] \underset{K}{*} \underset{\text{Poly}[K]}{\text{eval}}[\text{NevillePolynomial}[\langle x0, \overline{x}\rangle, \langle a0, \overline{a}\rangle, K], x0]\Big) \underset{K}{/} \Big(xn \underset{K}{-} x0\Big) = a0$$

which, using (Definition (Polynomial Domain)), is implied by:

(22) $$\underset{K}{0} \underset{K}{-} \Big((x0 - xn) \underset{K}{*} a0\Big) \underset{K}{/} \Big(xn \underset{K}{-} x0\Big) = a0.$$

Formula (22) can be verified by auxiliary knowledge on arithmetic in K. The remaining cases proceed analogously.

6 Application of the Algorithm to Concrete Examples

The recursive Algorithm["Neville"] can be used immediately in computations *without any translation* to some machine-executable language. The *Theorema* command "Compute" can perform rewriting using the recursive definition as given in Section 4. Rewriting is done by the interpreter of the underlying Mathematica system. In addition, it can access semantics for the algorithmic language constructs provided by the *Theorema* language (e.g. finite tuples, quantifiers with finite ranges, arithmetic on numbers, etc.), which is needed for performing polynomial arithmetic as defined in the polynomial functor in Section 2.

Compute[NevillePolynomial[$\langle 1, 2, 3, 4, 5\rangle$, $\langle 3, 1, 5, 2, 6\rangle$, $\mathbb{Q}$],
using → $\langle$Definition["Polynomial Domain"], Algorithm["Neville"]$\rangle$]

$$\left\langle 51, -\frac{1093}{12}, \frac{443}{8}, -\frac{161}{12}, \frac{9}{8}\right\rangle$$

This computation tells that the Neville-polynomial over $\mathbb{Q}$ for the tuples $\langle 1, 2, 3, 4, 5\rangle$ and $\langle 3, 1, 5, 2, 6\rangle$ is the polynomial $\langle 51, -\frac{1093}{12}, \frac{443}{8}, -\frac{161}{12}, \frac{9}{8}\rangle$, which would commonly be written as the arithmetic term $51 - \frac{1093}{12} x + \frac{443}{8} x^2 - \frac{161}{12} x^3 + \frac{9}{8} x^4$.

7 Conclusion

The *Theorema* system has been used in formal development of part of a mathematical theory. An entire exploration cycle—from defining mathematical concepts, over stating mathematical properties, computer-supported proving, until finally applying mathematical knowledge to concrete objects—has been carried through inside the system. The main objective of this case study is to show the integration of *proving* and *computing* in combination with an attractive mathematics-oriented syntax inside *one system*.

References

[AXIOM] Axiom. *Developed by IBM Research, directed by R. Jenks*. http://www.nag.com/symbolic_software.asp.

[Buchberger 96a] B. Buchberger: *Symbolic Computation: Computer Algebra and Logic*. In: Frontiers of Combining Systems (F. Baader, K.U. Schulz eds.), pp. 193-220. Applied Logic Series. Kluwer Academic Publishers, 1996.

[Buchberger 96b] B. Buchberger: *Mathematica as a Rewrite Language*. Invited paper in: Proceedings of the Fuji Conference on Functional Logic Programming, Shonan Village, Nov 1-4, 1996, (T. Ida ed.), pp. 1–13,Telos Publishing.

[MKM 03] Bruno Buchberger, Gaston Gonnet, Michiel Hazewinkel (eds.). *Mathematical Knowledgement Management*. Special issue of the journal Annals of Mathematics and Artificial Intelligence, Kluwer Publishing Company, 2003.

[Windsteiger 99] W. Windsteiger: *Building up Hierarchical Mathematical Domains Using Functors in Theorema*. In: A. Armando and T. Jebelean, editors, Electronic Notes in Theoretical Computer Science, vol 23-3, p. 83-102, Elsevier, 1999.

Some Grand Mathematical Challenges in Mechanized Mathematics

Jacques Calmet *

Institut for Algorithms and Cognitive Systems (IAKS)
University of Karlsruhe (TH)
calmet@ira.uka.de

Abstract. Grand mathematical challenges do exist in pure mathematics. Are some of the acute mathematical problems that we face when mechanizing mathematics true challenges? This short paper tries to assess through a few examples that they are indeed so.

1 Introduction

The quest for mathematically challenging problem is present in any field of science. A very recent example is from material science (Taylor [2003]). However, the most famous challenges concern Mathematics itself. A puzzling one was the proof of Fermat's theorem. It is barely necessary to cite the landmark presentation of David Hilbert at the 1900 Paris Conference. His list of problems for the 20th century was then extended to the well-known set of the 21 Hilbert's problems. On the eve of the 21th century, the International Mathematical Union asked a selected number of top mathematicians to contribute a similar list for the coming century. The contribution of Steve Smale has been widely distributed. It appeared first in the Mathematical Intelligencer (Smale [1998]). Two years later a version in French (Smale [2000]) appeared in the January issue of the Bulletin of the French Mathematical Society. It looks like Smale's list got a wide agreement and no other list of challenge problems has, apparently, been published. The three greatest open problems of mathematics are: the Riemann Hypothesis (Hilberth's 16th), Poincaré conjecture and "Does P=NP?". The latter is already tightly linked to our domain. One of the remaining challenging mathematical problems is very relevant to this community. It amounts to answer the question "What are the limits of (artificial or natural) intelligence?". This simple, apparently philosophical question leads in fact to very difficult mathematical problems such as the decidability of the Mandelbrot set.

To propose a definition of a mathematical challenge in theorem proving or in computer algebra that can be acknowledged by mathematicians is always touchy and sometimes impossible. However, if we introduce either of the words "mechanized" or "constructible" to qualify the part of Mathematics we deal with, we can then open a few tracks along the following directions.

* Work partially supported by the Calculemus Reserach Training Network, HPRN-CT-2000-00102

(i) Mechanize new areas of Mathematics such as algebraic topology or better Grothendieck's theory when also including geometry. Indefinite symbolic integration is a well-known example where a problem in analysis was turned into an algebraic problem. The methodology is constructive since Risch's algorithm decides whether or not integrability exists. It is also mechanized since the solution, when it does exist is constructed,

(ii) Identify and master new representations of mathematical objects. This is well understood when designing algebraic algorithms for computer algebra systems. A certainly challenging task is to investigate how algebraic fields (Laumon [1999]) could be introduced in mechanizing algebraic geometry problems. Also relevant are proof techniques in algebraic topology. This is a domain where the infinity plays a very special part compared to algebra, analysis, geometry of algebraic geometry. In these domains, the infinity is not really a challenge since proofs do not have to namely address this concept. In algebraic topology, the concept of infinity is very important when designing proofs,

(iii) Devise new proof techniques for domains where the amount of computation, not the theoretical difficulties is the challenge. An example is to prove some theorems on p-groups that would take a lifetime by hand calculation,

(iv) Space and time complexity issues when designing proofs and algorithms. Besides the "P=NP ?" problem already quoted, a prototypical example is the factorization of integer numbers. More practical examples arise when trying to improve doubly exponential algorithms such as the Gröbner bases algorithm, which play an important part in theorem proving in geometry, or when dealing with parameters as in constraint programming, which concerns any computing problem

(v) What means to prove? This looks like a silly question but a domain as demanding and "theoretical" as provable security sheds some light on this question.

The remaining part of this abstract is devoted to the presentation of some specific problems.

2 Provable security and proofs

The deduction community is much concerned by designing proof techniques for security protocols and there are many publications in this domain. It may be worthwhile to restrict the question to the simpler one "what is provable security?" and then to assess the part played by proofs in the answer. This is a domain of standard cryptography and a very nice state of the art is available in (Stern [2003]). Provable cryptography is an attempt to mathematically establish security. This is indeed very difficult and as a result, what is available is a form of "practical" provable security. It is possible to decompose provable security into 5 steps:

1. define the goal of the adversary,
2. define a security model,

3. provide a proof by reduction,
4. check the proof,
5. interpret the proof.

A first remark is that we are in engineering, not in mathematics but concepts are expressed mathematically. A second one is that provable security does not necessary yield proofs that are sound. What really matters, is that public key encryption cannot be broken. As a consequence it is not that surprising to notice that the 4th point on checking proofs may assert that a proof for an encryption mechanism is not correct. In fact "this does not matter so much". Indeed, there is usually enough time left, before the encryption is broken, to come out with the right proof.

An open but difficult problem is to extend provable security to security protocols. One may guess that this task implies to introduce a concept of randomness in the date structures and of probability in the proof techniques.

3 Involutive bases

Systems of polynomial equations are solved using Gröbner bases and the related Buchberger's algorithm. A Gröbner basis is simply a basis with "good" properties in a given ideal. Involutive bases are a very special kind of Gröbner bases with additional combinatorial properties that make them very useful for many applications (Calmet [2001]). They were first introduced by Janet a very long time ago. They exist in many polynomial algebras (also non-commutative ones) including ordinary polynomials and linear differential or difference operators. They are thus a possible approach to investigate symbolic solutions to system of (partial) differential equations. This is a domain where we need to find a suitable representation for differential objects. An overview is given in the final report of an INTAS project (Calmet [2002]).

On the theoretical side, numerous results on the relationships between different kinds of involutive bases, Gröbner bases and characteristic sets have been obtained both for ordinary and for differential ideals. Several characterisation theorems for involutive bases have been proven and the computation of (differential) dimension polynomials has been studied. We have thoroughly investigated the homological approach to involution via Spencer cohomology. An algebraic algorithm for the geometric completion to involution was developed (including a constructive solution of the problem of the so-called delta-regularity).

Although we label these results "theoretical", they are in fact pretty technical. Any of them require to establish existence and validity proofs. The leading idea is that when we identify the right representation, then proofs and thus the algorithms that are images of such proofs are much easier to discover. A much more challenging facet of what involutive bases are leading to is subsumed by the concept of global integrability.

4 Global integrability

Given a system of non-linear partial differential equations, can we decide of its integrability? A first answer is that we have tools, such as the Cartan-Kuranishi theorem, to decide of the local integrability but none to assess the global integrability. Physicists and mathematicians are investigating this problem for around 50 and 100 years respectively and no satisfactory solution is yet found.

A possible approach is to investigate the impact of involutive techniques outlined in the previous section in field theory, a domain of Physics. Most, if not all, physical models are represented by systems of partial differential equations. Among such systems are the well-known Yang-Mills or Einstein equations for instance. Without aiming at doing better than what the very many expert physicists of field theory are doing, it is possible to study whether some systems are integrable or not. What is challenging is to solve symbolically over- or under-determined systems of polynomial or differential equations or in simpler terms to extend the concept of Gröbner bases to such systems. This is again an old, well-known problem that was much earlier investigated by Cartan and his co-workers before being put aside. The need to design constructive methods in mechanized mathematics was at the origin of a revival. But, we still need to find out the proper representations in which to better formulate involutive bases. Again in very simple words, we are in a situation where we can get some information on local solutions of non-linear systems and we aim at extending them to some kind of non-local neighborhood. At this stage it is worthwhile to assess whether algebraic topology can be the key tool leading to a breakthrough in this domain. Physics texbooks such as (Weinberg [2002]) report that algebraic topology is already playing a role in obtaining approximate solutions to physical systems. This is a truly challenging problem where the challenge is to design constructive proofs and decision methods. This is a task better suited for computer scientists.

5 An example in group theory

This is a prototypical domain where some problems are more tedious and repetitive than difficult. Some proofs could take a lifetime to be completed by hand calculation. This area reminds of the beginning of computer algebra. The first successes that established the field were obtained in high energy particle physics, celestial mechanics or general relativity where repetitive, very long and tedious computations were required. Besides exceeding the human capability, they were also error prone when done on paper.

A test problem is as follows, where the word suitable is used to avoid a too long presentation of the problem:

> *Given a "suitable" infinite collection of p-groups, give a formula for the least n such that the i-th group in the collection can be embedded in S_n, not in S_{n-1}.*

This is, according to the experts, a very long term project even when coupling DSs and CASs. However, when analyzing the problem, it is possible to identify sub-problems. Many of them are purely computational ones. For instance, one must compute determinants of matrices. Depending on the size of these matrices, a very thorough management of the computation is required. There are deduction problems as well. One of them is supposed to be simple and can be seen as a test of feasibility. It is to check whether subgroups of the quaternion group of order 2^n are normal.

6 Conclusion

This selection of a few computational domains where a need for new proof techniques looks pretty obvious shows, hopefully, that we are facing some very challenging mathematical problems. Some may be qualified to be grand. The list here, as said by Smale in his paper, is only taken from problems where the author has some experience. This is a further proof that it ought to be easely enlarged. It is not explicitely mentioned in the section on global integrability that the required tools belong to algebraic topology. This is a domain where contacts to computer science are rather limited. Two pieces of works are worth citing. A first one is by Jesus Aransay at the University of La Rioja in Spain. It is on progress and performed within Calculemus. It deals with proving theorems in algebraic topology. A second one is the KENZO computer algebra system of F. Sergeraert at the Fourier Institute in Grenoble. It is the only computer algebra system enabling to perform computations in algebraic topology.

References

[2003] J. E. Taylor: Bulletin of the AMS, Vol. 40, No. 1, January 2003

[1998] S. Smale: Mathematical Problems for the Next Century, Mathematical Intelligencer Vol 20 No. 2, 7-15, 1998

[2000] S. Smale: Problèmes mathématiques pour le prochain siècle, Gazette des mathématiques, SMF, Janvier 2000.

[1999] G. Laumon and L. Moret-Bailly: "Champ algébriques",Vol. 39 in A Series of Modern Surveys in Mathematics, Springer, 1999

[2001] J. Calmet, M. Hausdorf and W. Seiler: A Constructive Introduction to Involution. Proceedings of ISACA2000 "Applications of Computer Algebra", R. Akerkar ed., Allied Publishers Limited, pp. 33-50, 2001

[2002] J. Calmet et al.: "INTAS - Final report", http://iaks-www.ira.uka.de/iaks-calmet/intas.htlm, 2002

[2003] J. Stern: Why Provable Security Matters?, In "Advances in Cryptology ", Proceedings of Eurocrypt 2003, Warsaw, May 2003, Biham E., Ed., LNCS 2656 , Springer, 2003

[2002] S. Weinberg: The Quantum Theory of Fields, Volume II, Chapter 23, Cambridge University Press, 1996