

Johan Jeuring John A. Campbell
Jacques Carette Gabriel Dos Reis
Petr Sojka Makarius Wenzel
Volker Sorge (Eds.)

LNAI 7362

Intelligent Computer Mathematics

11th International Conference, AISC 2012
19th Symposium, Calculemus 2012
5th International Workshop, DML 2012
11th International Conference, MKM 2012
Systems and Projects, Held as Part of CICM 2012
Bremen, Germany, July 2012, Proceedings

Lecture Notes in Artificial Intelligence 7362

Subseries of Lecture Notes in Computer Science

LNAI Series Editors

Randy Goebel

University of Alberta, Edmonton, Canada

Yuzuru Tanaka

Hokkaido University, Sapporo, Japan

Wolfgang Wahlster

DFKI and Saarland University, Saarbrücken, Germany

LNAI Founding Series Editor

Joerg Siekmann

DFKI and Saarland University, Saarbrücken, Germany

Johan Jeuring John A. Campbell
Jacques Carette Gabriel Dos Reis
Petr Sojka Makarius Wenzel
Volker Sorge (Eds.)

Intelligent Computer Mathematics

11th International Conference, AISC 2012
19th Symposium, Calculemus 2012
5th International Workshop, DML 2012
11th International Conference, MKM 2012
Systems and Projects, Held as Part of CICM 2012
Bremen, Germany, July 8-13, 2012, Proceedings

Series Editors

Randy Goebel, University of Alberta, Edmonton, Canada
Jörg Siekmann, University of Saarland, Saarbrücken, Germany
Wolfgang Wahlster, DFKI and University of Saarland, Saarbrücken, Germany

Volume Editors

Johan Jeuring
Utrecht University, The Netherlands, E-mail: j.t.jeuring@uu.nl

John A. Campbell
University College London, UK, E-mail: j.campbell@cs.ucl.ac.uk

Jacques Carette
McMaster University, Hamilton, ON, Canada, E-mail: carette@mcmaster.ca

Gabriel Dos Reis
Texas A&M University, College Station, TX, USA, E-mail: gdr@cs.tamu.edu

Petr Sojka
Masaryk University, Brno, Czech Republic, E-mail: sojka@fi.muni.cz

Makarius Wenzel
Université de Paris-Sud, Orsay Cedex, France, E-mail: makarius.wenzel@lri.fr

Volker Sorge
The University of Birmingham, UK, E-mail: v.sorge@cs.bham.ac.uk

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-31373-8 e-ISBN 978-3-642-31374-5
DOI 10.1007/978-3-642-31374-5
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012940388

CR Subject Classification (1998): I.1, F.4.1, I.2.2-3, I.2.6, I.2, F.3.1, D.2.4, F.3, H.3.7, H.3, G.4, H.2.8

LNCS Sublibrary: SL 7 – Artificial Intelligence

© Springer-Verlag Berlin Heidelberg 2012

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

As computers and communications technology advance, greater opportunities arise for intelligent mathematical computation. While computer algebra, automated deduction, mathematical publishing and novel user interfaces individually have long and successful histories, we are now seeing increasing opportunities for synergy among these areas. The series of Conferences on Intelligent Computer Mathematics (CICM) hosts collections of co-located meetings, allowing researchers and practitioners active in these related areas to share recent results and identify the next challenges.

The fifth in this series of Conferences on Intelligent Computer Mathematics was held in Bremen, Germany, in 2012. Previous conferences, all also published in Springer's *Lecture Notes in Artificial Intelligence* series, were held in the UK (Birmingham, 2008: LNAI 5144), Canada (Grand Bend, Ontario, 2009: LNAI 5625), France (Paris, 2010: LNAI 6167) and Italy (Bertinoro, 2011: LNAI 6824). CICM 2012 included four long-standing international meetings:

- 11th International Conference on Mathematical Knowledge Management (MKM 2012)
- 19th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning (Calculemus 2012)
- 11th International Conference on Artificial Intelligence and Symbolic Computation (AISC 2012)
- 5th Workshop/Conference on Digital Mathematics Libraries (DML 2012)

Since 2011, CICM also offers a track for brief descriptions of systems and projects that span the MKM, Calculemus, AISC, and DML topics, the “Systems and Projects” track. The proceedings of the four international meetings and the Systems and Projects track are collected in this volume.

CICM 2012 also contained the following activities:

- Demonstrations of the systems presented in the Systems and Projects track
- Less formal “work in progress” sessions

We used the “multi-track” features of the EasyChair system, and our thanks are due to Andrei Voronkov and his team for this and many other features. The multi-track feature also allowed transparent handling of conflicts of interest between the Track Chairs and submissions: these submissions were moved to a separate track overseen by the General Chair. There were 60 submissions, eight of which were withdrawn. Each of the remaining 52 submission was reviewed by at least two, and on average three, Program Committee members. The committee decided to accept 38 papers. However, this is a conflation of tracks with different acceptance characteristics. The track-based acceptance rates were:

MKM	13 acceptances out of 19 submissions
Calculemus	6 acceptances out of 9 submissions
AISC	6 acceptances out of 8 submissions
DML	2 acceptances out of 3 submissions
S & P	11 acceptances out of 12 submissions

One paper was not submitted to a particular track, and was rejected.

Three invited talks were given. The first one was by Conor McBride from the Department of Computer and Information Sciences, University of Strathclyde, and was entitled “A Prospection for Reflection”:

Gödel’s incompleteness theorems tell us that there are effective limitations on the capacity of logical systems to admit reasoning about themselves. However, there are solid pragmatic reasons for wanting to try: we can benefit considerably by demonstrating that systematic patterns of reasoning (and programming, of course) are admissible. It is very useful to treat goals as data in order to attack them with computation, adding certified automation to interactive proof tools, delivering the efficiency required to solve compute-intensive problems with no loss of trust.

Dependent type theory provides a ready means of reflection: goals become types, and types may be computed from data. This technique has proven highly successful when tightly targeted on specific problem domains. But we may yet ask the bold question of how large a universe of problems we can effectively reflect: how much of our type theory can we encode within its own notion of data? To what extent can our type theory capture its own typing discipline? Might we construct a hierarchy of type theories where the whole of each lower level can be reflected at higher levels? In this talk, I shall outline grounds for modest optimism and propose a plan of campaign. The obstacles may turn out to be as fascinating as the objective. The reward, if we can reach it, is a flexible technology for certified automation in problem-solving, honestly articulating what at least computers can do.

The second invited talk was by Cezar Ionescu from the Potsdam Institute for Climate Impact Research, on “Increasingly Correct Scientific Programming”:

Dependently typed languages promise an elegant environment for programming from specifications: the properties that a program should satisfy are expressed as logical formulas and encoded via the Curry–Howard isomorphism as a type, a candidate implementation should be a member of this type, and the type checker verifies whether this is indeed the case. But sometimes the type checker demands “too much”: in particular, in scientific programming, it often seems that one must formalize all of real analysis before writing a single line of useful code. Alternatively, one can use mechanisms provided by the language in order to circumvent the type checker, and confirm that “real programmers can write Fortran in any language.” We present an example of navigating between these

extremes in the case of economic modeling. First, we use postulates in order to be able to reuse code, and achieve a kind of conditional correctness (for example, we can find a Walrasian equilibrium if we are able to solve a convex optimization problem). We then remove more and more of the postulates, replacing them with proofs of correctness, by using interval arithmetic methods.

Finally, Yannis Haralambous, Département Informatique, Télécom Bretagne, gave a talk on “Text Mining Methods Applied to Mathematical Texts.”

April 2012

Johan Jeuring
John A. Campbell
Jacques Carette
Gabriel Dos Reis
Petr Sojka
Makarius Wenzel
Volker Sorge

Organization

CICM 2012 was organized by the Conference on Intelligent Computer Mathematics Steering Committee, which was formed at CICM 2010 as a parent body to the long-standing Calculemus and Mathematical Knowledge Management special interest groups. The conferences organized by these interest groups continue as special tracks in the CICM conference. The AISC conference, which is only organized every other year, and DML workshop were organized in 2012 too. These tracks and the Systems and Projects track had independent Track Chairs and Program Committees. Local arrangements, the life-blood of any conference, were handled by the Department of Computer Science of the Jacobs University Bremen, Germany, and DFKI, Bremen, Germany.

CICM Steering Committee

Secretary

Michael Kohlhase Jacobs University Bremen, Germany

Calculemus Delegate

Renaud Rioboo ENSIIE, France

Treasurer

William Farmer McMaster University, Canada

DML Delegate

Thierry Bouche Université Joseph Fourier Grenoble, France

CICM PC Chair 2011

James Davenport University of Bath, UK

CICM PC Chair 2012

Johan Jeuring Utrecht University and Open University,
The Netherlands

MKM Trustees

Serge Autexier	Florian Rabe	Alan Sexton
James Davenport	Claudio Sacerdoti Coen	Makarius Wenzel
Patrick Ion (Treasurer)		

Calculemus Trustees

David Delahaye
Gabriel Dos Reis
William Farmer

Paul Jackson
Renaud Rioboo
Volker Sorge

Stephen Watt
Makarius Wenzel

AISC Steering Committee

Serge Autexier
Jacques Calmet

John Campbell
Jacques Carette

Eugenio Roanes-Lozano
Volker Sorge

CICM 2012 Officers

General Program Chair

Johan Jeuring

Utrecht University and Open University,
The Netherlands

Local Arrangements

Michael Kohlhase
Serge Autexier

Jacobs University Bremen, Germany
DFKI, Germany

MKM Track Chair

Makarius Wenzel

LRI, Paris Sud, France

Calculemus Track Chair

Gabriel Dos Reis

Texas A&M University, USA

AISC Track CoChairs

John A. Campbell
Jacques Carette

University College London, UK
McMaster University, Canada

DML Track Chair

Petr Sojka

Masaryk University Brno, Czech Republic

S & P Track Chair

Volker Sorge

University of Birmingham, UK

Program Committee Mathematical Knowledge Management

David Aspinall	University of Edinburgh, UK
Jeremy Avigad	Carnegie Mellon University, USA
Mateja Jamnik	University of Cambridge, UK
Cezary Kaliszyk	University of Tsukuba, Japan
Manfred Kerber	University of Birmingham, UK
Christoph Lüth	DFKI, Germany
Adam Naumowicz	University of Białystok, Poland
Jim Pitman	University of California at Berkeley, USA
Pedro Quaresma	University of Coimbra, Portugal
Florian Rabe	Jacobs University Bremen, Germany
Claudio Sacerdoti Coen	University of Bologna, Italy
Enrico Tassi	INRIA Saclay, France
Makarius Wenzel	LRI, Paris Sud, France
Freek Wiedijk	Radboud University Nijmegen, The Netherlands

Program Committee Calculemus

Andrea Asperti	University of Bologna, Italy
Laurent Bernardin	Maplesoft, Canada
James H. Davenport	University of Bath, UK
Gabriel Dos Reis	Texas A&M University, USA
Ruben Gamboa	University of Wyoming, USA
Mark Giesbrecht	University of Waterloo, Canada
Sumit Gulwani	Microsoft Research, USA
John Harrison	Intel Corporation, USA
Joris van der Hoeven	CNRS, Ecole Polytechnique, France
Hoon Hong	North Carolina State University, USA
Loïc Pottier	INRIA Sophia-Antipolis, France
Wolfgang Windsteiger	RISC Linz, Johannes Kepler University, Austria

Program Committee Artificial Intelligence and Symbolic Computation

Serge Autexier	DFKI, Germany
Jacques Calmet	Karlsruhe Institute of Technology, Germany
John Campbell	University College London, UK
Jacques Carette	McMaster University, Canada
Simon Colton	Imperial College London, UK
Jacques Fleuriot	University of Edinburgh, UK

Andrea Kohlhase	Jacobs University Bremen, Germany
Taisuke Sato	Tokyo Institute of Technology, Japan
Erik Postma	Maplesoft, Canada
Alan Sexton	Birmingham University, UK
Chung-chieh Shan	Cornell University, USA
Toby Walsh	University of New South Wales, Australia
Stephen Watt	University of Western Ontario, Canada

Program Committee Digital Mathematics Libraries

José Borbinha	Technical University of Lisbon, Portugal
Thierry Bouche	Université Joseph Fourier Grenoble, France
Michael Doob	University of Manitoba, Canada
Thomas Fischer	Goettingen University, Germany
Yannis Haralambous	Télécom Bretagne, France
Václav Hlaváč	Czech Technical University Prague, Czech Republic
Michael Kohlhase	Jacobs University Bremen, Germany
Janka Chlebíková	University of Portsmouth, UK
Enrique Maciás-Virgós	University of Santiago de Compostela, Spain
Bruce Miller	NIST, USA
Jiří Rákosník	Mathematical Institute Prague, Czech Republic
Eugénio A.M. Rocha	University of Aveiro, Portugal
David Ruddy	Cornell University, USA
Volker Sorge	University of Birmingham, UK
Petr Sojka	Masaryk University Brno, Czech Republic
Masakazu Suzuki	Kyushu University, Japan

Program Committee Systems and Projects

Josef Baker	University of Birmingham, UK
John Charnley	Imperial College London, UK
Manuel Kauers	RISC Linz, Johannes Kepler University, Austria
Koji Nakagawa	Kyushu University, Japan
Christoph Lange	Jacobs University Bremen, Germany
Piotr Rudnicki	University of Alberta, Canada
Volker Sorge	University of Birmingham, UK
Josef Urban	Radboud University Nijmegen, The Netherlands
Richard Zanibbi	Rochester Institute of Technology, USA

Additional Referees

In addition to the many members of the Program Committees who refereed for other tracks, we are grateful to the following additional referees.

Marc Bezem	Paul Libbrecht	Carst Tankink
Flaminia Cavallo	Petros Papapanagiotou	René Thiemann
Dominik Dietrich	Adam Pease	Matej Urbas
Holger Gast	Wolfgang Schreiner	Jiří Vyskočil
Mihnea Iancu	Christian Sternagel	Iain Whiteside
Temur Kutsia	Geoff Sutcliffe	

Sponsoring Institutions

Jacobs University Bremen and DFKI, Bremen, Germany

Table of Contents

Mathematical Knowledge Management 2012

Dependencies in Formal Mathematics: Applications and Extraction for Coq and Mizar	1
<i>Jesse Alama, Lionel Mamane, and Josef Urban</i>	
Proof, Message and Certificate	17
<i>Andrea Asperti</i>	
Challenges and Experiences in Managing Large-Scale Proofs	32
<i>Timothy Bourke, Matthias Daum, Gerwin Klein, and Rafal Kolanski</i>	
Semantic Alliance: A Framework for Semantic Allies	49
<i>Catalin David, Constantin Jucovski, Andrea Kohlhase, and Michael Kohlhase</i>	
Extending MKM Formats at the Statement Level	65
<i>Fulya Horozal, Michael Kohlhase, and Florian Rabe</i>	
A Streaming Digital Ink Framework for Multi-party Collaboration	81
<i>Rui Hu, Vadim Mazalov, and Stephen M. Watt</i>	
Cost-Effective Integration of MKM Semantic Services into Editing Environments	96
<i>Constantin Jucovski</i>	
Understanding the Learners' Actions when Using Mathematics Learning Tools	111
<i>Paul Libbrecht, Sandra Rebholz, Daniel Herding, Wolfgang Müller, and Felix Tscheulin</i>	
Towards Understanding Triangle Construction Problems	127
<i>Vesna Marinković and Predrag Janičić</i>	
A Query Language for Formal Mathematical Libraries	143
<i>Florian Rabe</i>	
Abramowitz and Stegun – A Resource for Mathematical Document Analysis	159
<i>Alan P. Sexton</i>	
Point-and-Write – Documenting Formal Mathematics by Reference	169
<i>Carst Tankink, Christoph Lange, and Josef Urban</i>	

An Essence of SSReflect	186
<i>Iain Whiteside, David Aspinall, and Gudmund Grov</i>	

Calcuemus 2012

Theory Presentation Combinators	202
<i>Jacques Carette and Russell O'Connor</i>	
Verifying an Algorithm Computing Discrete Vector Fields for Digital Imaging	216
<i>Jónathan Heras, María Poza, and Julio Rubio</i>	
Towards the Formal Specification and Verification of Maple Programs	231
<i>Muhammad Taimoor Khan and Wolfgang Schreiner</i>	
Formalizing Frankl's Conjecture: FC-Families	248
<i>Filip Marić, Miodrag Živković, and Bojan Vučković</i>	
CDCL-Based Abstract State Transition System for Coherent Logic	264
<i>Mladen Nikolić and Predrag Janičić</i>	
Speeding Up Cylindrical Algebraic Decomposition by Gröbner Bases ...	280
<i>David J. Wilson, Russell J. Bradford, and James H. Davenport</i>	

Artificial Intelligence and Symbolic Computation 2012

A System for Axiomatic Programming	295
<i>Gabriel Dos Reis</i>	
Reasoning on Schemata of Formulæ	310
<i>Mnacho Echenim and Nicolas Peltier</i>	
Management of Change in Declarative Languages	326
<i>Mihnea Iancu and Florian Rabe</i>	
MathWebSearch 0.5: Scaling an Open Formula Search Engine	342
<i>Michael Kohlhase, Bogdan A. Matican, and Corneliu-Claudiu Prodescu</i>	
Real Algebraic Strategies for MetiTarski Proofs	358
<i>Grant Olney Passmore, Lawrence C. Paulson, and Leonardo de Moura</i>	

A Combinator Language for Theorem Discovery	371
<i>Phil Scott and Jacques Fleuriot</i>	

Digital Mathematics Libraries 2012

DynGenPar – A Dynamic Generalized Parser for Common Mathematical Language	386
<i>Kevin Kofler and Arnold Neumaier</i>	

Writing on Clouds	402
<i>Vadim Mazalov and Stephen M. Watt</i>	

Systems and Projects 2012

A Web Interface for Matita	417
<i>Andrea Asperti and Wilmer Ricciotti</i>	

MaxTract: Converting PDF to L ^A T _E X, MathML and Text	422
<i>Josef B. Baker, Alan P. Sexton, and Volker Sorge</i>	

New Developments in Parsing Mizar	427
<i>Czesław Byliński and Jesse Alama</i>	

Open Geometry Textbook: A Case Study of Knowledge Acquisition via Collective Intelligence (Project Description)	432
<i>Xiaoyu Chen, Wei Li, Jie Luo, and Dongming Wang</i>	

Project Presentation: Algorithmic Structuring and Compression of Proofs (ASCOP)	438
<i>Stefan Hetzl</i>	

On Formal Specification of Maple Programs	443
<i>Muhammad Taimoor Khan and Wolfgang Schreiner</i>	

The PLANETARY Project: Towards eMath3.0	448
<i>Michael Kohlhase</i>	

Tentative Experiments with Ellipsis in Mizar	453
<i>Artur Kornilowicz</i>	

Reimplementing the Mathematics Subject Classification (MSC) as a Linked Open Dataset	458
<i>Christoph Lange, Patrick Ion, Anastasia Dimou, Charalampos Bratsas, Joseph Corneli, Wolfram Sperber, Michael Kohlhase, and Ioannis Antoniou</i>	

The Distributed Ontology Language (DOL): Ontology Integration and Interoperability Applied to Mathematical Formalization	463
<i>Christoph Lange, Oliver Kutz, Till Mossakowski, and Michael Grüninger</i>	
Isabelle/jEdit – A Prover IDE within the PIDE Framework	468
<i>Makarius Wenzel</i>	
Author Index	473

Dependencies in Formal Mathematics: Applications and Extraction for Coq and Mizar

Jesse Alama^{1,*}, Lionel Mamane^{2,**}, and Josef Urban^{3,***}

¹ Center for Artificial Intelligence
New University of Lisbon
j.alama@fct.unl.pt

<http://centria.di.fct.unl.pt/~alama/>

² 59, rue du X Octobre
L-7243 Bereldange
Luxembourg
lionel@mamane.lu

³ Institute for Computing and Information Sciences
Radboud University Nijmegen
josef.urban@gmail.com

Abstract. Two methods for extracting detailed formal dependencies from the Coq and Mizar system are presented and compared. The methods are used for dependency extraction from two large mathematical repositories: the Coq Repository at Nijmegen and the Mizar Mathematical Library. Several applications of the detailed dependency analysis are described and proposed. Motivated by the different applications, we discuss the various kinds of dependencies that we are interested in, and the suitability of various dependency extraction methods.

1 Introduction

This paper presents two methods for extracting detailed formal dependencies from two state-of-the-art interactive theorem provers (ITPs) for mathematics: the Coq system and the Mizar system. Our motivation for dependency extraction is application-driven. We are interested in using detailed dependencies for fast refactoring of large mathematical libraries and wikis, for AI methods in automated reasoning that learn from previous proofs, for improved interactive editing of formal mathematics, and for foundational research over formal mathematical libraries.

* Supported by the ESF research project *Dialogical Foundations of Semantics* within the ESF Eurocores program *LogICCC* (funded by the Portuguese Science Foundation, FCT LogICCC/0001/2007). Research for this paper was partially done while a visiting fellow at the Isaac Newton Institute for the Mathematical Sciences in the program ‘Semantics & Syntax’.

** Supported during part of the research presented here by the NWO project “Formal Interactive Mathematical Document: Creation and Presentation”; during that time, he was affiliated with the ICIS, Radboud University Nijmegen.

*** Supported by the NWO project “MathWiki: A Web-based Collaborative Authoring Environment for Formal Proofs”.

These applications require different notions of *formal dependency*. We discuss these different requirements, and as a result provide implementations that in several important aspects significantly differ from previous methods. For Mizar, the developed method captures practically all dependencies needed for successful re-verification of a particular formal text (i.e., also notational dependencies, automations used, etc.), and the method attempts hard to determine the minimal set of such dependencies. For Coq, the method goes farther towards re-verification of formal texts than previous methods [5,13,4] that relied solely on the final proof terms. For example, we can already track Coq dependencies that appear during the tactic interpretation, but that do not end up being used in the final proof term.

The paper is organized as follows. Section 2 briefly discusses the notion of formal dependency. Section 3 describes the implementation of dependency extraction in the Coq system, and Section 4 describes the implementation in the Mizar system. Section 5 compares the two implemented approaches to dependency computation. Section 6 describes several experiments and measurements conducted using our implementations on the CoRN and MML libraries, including training of AI/ATP proof assistance systems on the data, and estimating the speed-up for collaborative large-library developments. Section 8 concludes.

2 Dependencies: What Depends on What?

Generally, we say that a definition, or a theorem, T *depends* on some definition, lemma or other theorem T' , (or equivalently, that T' is a *dependency* of T) if T “needs” T' to exist or hold. The main way such a “need” arises is that the well-formedness, justification, or provability of T does not hold in the absence of T' . We consider formal mathematics done in a concrete proof assistant so we consider mathematical and logical constructs not only as abstract entities depending on each other, but also as concrete objects (e.g., texts, syntax trees, etc.) in the proof assistants. For our applications, there are at least two different notions of “dependency” we are interested in:

- Semantic/logical: One might claim, for example, that in the Coq context a λ -term (or proof object in the underlying formal framework) contains all dependencies of interest for a particular theorem, regardless of any notational conventions, library mechanisms, etc.
- Pragmatic: Such dependencies are met if a particular item still compiles, regardless of possibly changed underlying semantics. This view takes the whole proof assistant as the locus of dependency, with its sophisticated mechanisms such as auto hint databases, notations, type automations, definitions expansions, proof search depth, parser settings, hidden arguments, etc.

Formal dependencies can also be implicit and explicit. In the simple world of first-order automated theorem proving, proofs and their dependencies are generally quite detailed and explicit about (essentially) all logical steps, even very small ones (such as the steps taken in a resolution proof). But in ITPs, which are

generally oriented toward human mathematicians, one of the goals is to allow the users to express themselves with minimal logical verbosity and ITPs come with a number of implicit mechanisms. Examples are type mechanisms (e.g., type-class automations of various flavors in `Coq` [14] and `Isabelle` [8], Prolog-like types in `Mizar` [17,15]), hint mechanisms (in `Coq` and `Isabelle`), etc. If we are interested in giving a complete answer to the question of what a formalized proof depends upon, we must expose such implicit facts and inferences.

Formal dependencies reported by ITPs are typically *sufficient*. Depending on the extraction mechanism, redundant dependencies can be reported. Bottom-up procedures like congruence-closure and type closure in `Mizar` (and e.g., type-class mechanisms in other ITPs) are examples of mechanisms when the ITP uses available knowledge exhaustively, often drawing in many *unnecessary* dependencies from the context. For applications, it is obviously better if such unnecessary dependencies can be removed .

3 Dependency Extraction in `Coq`

Recall that `Coq` is based on the Curry-Howard isomorphism, meaning that:

1. A statement (formula) is encoded as a type.
2. There is, at the “bare” logical level, no essential difference between a definition and a theorem: they are both the binding (in the environment) of a name to a type (type of the definition, statement of the theorem) and a term (body of the definition, proof of the theorem).
3. Similarly, there is no essential difference between an axiom and a parameter: they are both the binding (in the environment) of a name to a type (statement of the axiom, type of the parameter, e.g. “natural number”).
4. There is, as far as `Coq` is concerned, no difference between the notions of theorem, lemma, corollary, ...

The type theory implemented by `Coq` is called the predicative calculus of inductive constructions, abbreviated as pCIC.

There are essentially three groups of `Coq` commands that need to be treated by the dependency tracking:¹

1. Commands that register a new logical construct (definition or axiom), either
 - From scratch. That is, commands that take as arguments a name and a type and/or a body, and that add the definition binding this name to this type and/or body. The canonical examples are

Definition Name : type := body

and

Axiom Name : type

The type can also be given implicitly as the inferred type of the body, as in

¹ As far as logical constructs are concerned.

```
Definition Name := body
```

- Saving the current (completely proven) theorem in the environment. These are the “end of proof” commands, such as `Qed`, `Save`, `Defined`.
2. Commands that make progress in the current proof, which is necessarily made in several steps:
 - (a) Opening a new theorem, as in

```
Theorem Name : type
```

or

```
Definition Name : type
```

- (b) An arbitrary strictly positive amount of proof steps.
 - (c) Saving that theorem in the environment. These commands update (by adding exactly *one* node) the internal `Coq` structure called “proof tree”.
3. Commands that open a new theorem, that will be proven in multiple steps.

The dependency tracking is implemented as suitable hooks in the `Coq` functions that the three kinds of commands eventually call. When a new construct is registered in the environment, the dependency tracking walks over the type and body (if present) of the new construct and collects all constructs that are referenced. When a proof tree is updated, the dependency tracking examines the top node of the new proof tree (note that this is always the only change with regards to the previous proof tree). The commands that update the proof tree (that is, make a step in the current proof) are called `tactics`. `Coq`’s tactic interpretation goes through three main phases:

1. parsing;
2. Ltac² expansion;
3. evaluation.

The tactic structure after each of these phases is stored in the proof tree. This allows to collect all construct references mentioned at any of these tree levels. For example, if tactic `Foo T` is defined as

```
try apply BolzanoWeierstrass;
solve [ T | auto ]
```

and the user invokes the tactic as `Foo FeitThompson`, then the first level will contain (in parsed form) `Foo FeitThompson`, the second level will contain (in parsed form)

```
try apply BolzanoWeierstrass;
solve [ FeitThompson | auto ].}
```

and the third level can contain any of:

² Ltac is the `Coq`’s tactical language, used to combine tactics and add new user-defined tactics.

- `refine (BolzanoWeierstrass ...)`,
- `refine (FeitThompson ...)`,
- something else, if the proof was found by `auto`.

The third level typically contains only a few of the basic atomic fundamental rules (tactics) applications, such as `refine`, `intro`, `rename` or `convert`, and combinations thereof.

3.1 Dependency Availability, Format, and Protocol

Coq supports several interaction protocols: the `coqtop`, `emacs` and `coq-interface` protocols. Dependency tracking is available in the program implementing the `coq-interface` protocol which is designed for machine interaction. The dependency information is printed in a special message for each *potentially progress-making command* that can give rise to a dependency.³ A *potentially progress-making command* is one whose purpose is to change Coq’s state. For example, the command `Print Foo`, which displays the previously loaded mathematical construct `Foo`, is not a potentially progress-making command.⁴ Any tactic invocation is a potentially progress-making command. For example, the tactic `auto` silently succeeds (without any effect) if it does not completely solve the goal it is assigned to solve. In that case, although that particular invocation did not make any actual progress in the proof, `auto` is still considered a potentially progress-making command, and the dependency tracking outputs the message ‘`dependencies: (empty list)`’. Other kinds of progress-making commands include, for example notation declarations and morphisms registrations. Some commands, although they change Coq’s state, might not give rise to a dependency. For example, the `Set Firstorder Depth` command, taking only an integer argument, changes the maximum depth at which the `firstorder` tactic will search for a proof. For such a command, no dependency message is output.

One command may give rise to several dependency messages, when they change Coq’s state in several different ways. For example, the `intuition` tactic⁵ can, mainly for efficiency reasons, construct an ad hoc lemma, register it into the global environment and then use that lemma to prove the goal it has been assigned to solve, instead of introducing the ad hoc lemma as a local hypothesis through a cut. This is mainly an optimization: The ad hoc lemma is defined as

³ In other words, the system gives dependencies of individual *proof steps*, not only of whole proofs.

⁴ Thus, although this commands obviously needs item `Foo` to be defined to succeed, the dependency tracking does not output that information. That is not a problem in practice because such commands are usually issued by a user interface to treat an interactive user request (e.g. “show me item `Foo`”), but are not saved into the script that is saved on disk. Even if they were saved into the script, adding or removing them to (from, respectively) the script does not change the semantics of the script.

⁵ The intuition tactic is a decision procedure for intuitionistic propositional calculus based on the contraction-free sequent calculi LJT* of Roy Dyckhof, extended to hand over subgoals which it cannot solve to another tactic.

“opaque”, meaning that the typechecking (proofchecking) algorithm is not allowed to unfold the body (proof) of the lemma when the lemma is invoked, and thus won’t spend any time doing so. By contrast, a local hypothesis is always “transparent”, and the typechecking algorithm is allowed to unfold its body. For the purpose of dependency tracking this means that `intuition` makes *two* conceptually different steps:

1. register a new global lemma, under a fresh name;
2. solve the current subgoal in the proof currently in progress.

Each of these steps gives rise to different dependencies. For example, if the current proof is `BolzanoWeierstrass`, then the new global lemma gives rise to dependencies of the form

“`BolzanoWeierstrass_subproofN` depends on ...”

where the `_subproofN` suffix is Coq’s way of generating a fresh name. Closing of the subgoal by use of `BolzanoWeierstrass_subproofN` then gives rise to the dependency

“`BolzanoWeierstrass` depends on `BolzanoWeierstrass_subproofN`”

3.2 Coverage and Limitations

The Coq dependency tracking is already quite extensive, and sufficient for the whole Nijmegen CoRN corpus. Some restrictions remain in parts of the Coq internals that the second author does not yet fully understand.⁶ Our interests (and experiments) include not only purely mathematical dependencies that can be found in the proof terms (for previous work see also [13,4]), but also fast recompilation modes for easy authoring of formal mathematics in large libraries and formal wikis. The Coq dependency tracking code currently finds all logically relevant dependencies from the proof terms, even those that arise from automation tactics. It does not handle yet the non-logical dependencies. Examples include notation declarations, morphism and equivalence relation declarations,⁷ `auto` hint database registrations,⁸ but also tactic interpretation. At this stage, we don’t handle most of these, but as already explained, the internal structure of Coq lends itself well to collecting dependencies that appear at the various levels of tactic interpretation. This means that we can already handle the (*non-semantic*) dependencies on logical constructs that appear during the tactic interpretation, but that do not end up being used in the final proof term.

⁶ E.g. a complete overview of all datatypes of tactic arguments and how “dynamics” are used in tactic expressions.

⁷ So that the tactics for equality can handle one’s user-defined equality.

⁸ `auto` not only needs that the necessary lemmas be available in the environment, but it also needs to be specifically instructed to try to use them, through a mechanism where the lemmas are registered in a “hint database”. Each invocation of `auto` can specify which hint databases to use.

Some of the non-logical dependencies are a more difficult issue in practice, albeit not always in theory. For example, several dependencies related to tactic parametrization (`auto` hint databases, `firstorder` proof depth search, morphism declarations) need specific knowledge of how the tactic is influenced by parameters or previous non-logical declarations. The best approach to handle such dependencies seems to be to change (at the OCaml source level in Coq) the type of a tactic, so that the tactic itself is responsible for providing such dependencies. This will however have to be validated in practice, provided that we manage to persuade the greater Coq community about the importance and practical usefulness of complete dependency tracking for formal mathematics and for research based on it.

Coq also presents an interesting corner case as far as opacity of dependencies is concerned. On the one hand, Coq has an explicit management of opacity of items; an item originally declared as opaque can only be used generically with regards to its type; no information arising from its body can be used, the only information available to other items is the type. Lemmas and theorems are usually declared opaque⁹, and definitions usually declared transparent, but this is not forced by the system. In some cases, applications of lemmas need to be transparent. Coq provides an easy way to decide whether a dependency is opaque or transparent: dependencies on opaque objects can only be opaque, and dependencies on transparent objects are to be considered transparent.

Note that the pCIC uses a universe level structure, where the universes have to be ordered in a well-founded way at all times. However, the ordering constraints between the universes are hidden from the user, and are absent from the types (statements) the user writes. Changing the proof of a theorem T can potentially have an influence on the universe constraints of the theorem. Thus, changing the body of an opaque item T' appearing in the proof of T can change the universe constraints attached to it, potentially in a way that is incompatible with the way it was previously used in the body of T . Detecting whether the universe constraints have changed or not is not completely straightforward, and needs specific knowledge of the pCIC. But unless one does so, for complete certainty of correctness of the library as a whole, one has to consider *all* dependencies as transparent. Note that in practice universe constraint incompatibilities are quite rare. A large library may thus optimize its rechecking after a small change, and not immediately follow opaque reverse dependencies. Instead, fully correct universe constraint checking could be done in a postponed way, for example by rechecking the whole library from scratch once per week or per month.

4 Dependency Extraction in Mizar

Dependency computation in Mizar differs from the implementation provided for Coq, being in some sense much simpler, but at the same time also more robust with respect to the potential future changes of the Mizar codebase. For comparison of the techniques, see Section 5. For a more detailed discussion of Mizar, see [11] or [7].

⁹ Thereby following the mathematical principle of **proof irrelevance**.

In Mizar, every article A has its own environment \mathcal{E}_A specifying the context (theorems, definitions, notations, etc.) that is used to verify the article. \mathcal{E}_A , is usually a rather conservative overestimate of the items that the article actually needs. For example, even if an article A needs only one definition (or theorem, or notation, or scheme, or . . .) from article B , all the definitions (theorems, notations, schemes, . . .) from B will be present in \mathcal{E}_A . The *dependencies for an article* A are computed as the smallest environment \mathcal{E}'_A under which A is still Mizar-verifiable (and has the same semantics as A did under \mathcal{E}_A). To get dependencies of a particular Mizar item I (theorem, definition, etc.), we first create a *microarticle* containing essentially just the item I , and compute the dependencies of this microarticle.

More precisely, computing fine-grained dependencies in Mizar takes three steps:

Normalization. Rewrite every article of the Mizar Mathematical Library so that:

- Each definition block defines exactly one concept.
(Definition blocks that contain multiple definitions or notations can lead to false positive dependencies.)
- All toplevel logical linking is replaced by explicit reference: constructions such as

```
ϕ; then ψ;
```

whereby the statement ψ is justified by the statement ϕ , are replaced by

```
Label1: ϕ;
Label2: ψ by Label1;
```

where `Label1` and `Label2` are new labels. By doing this transformation, we ensure that the only way that a statement is justified by another is through explicit reference.

- Segments of reserved variables all have length exactly 1. For example, constructions such as

```
reserve A for set ,
        B for non empty set ,
        f for Function of A, B,
        M for Cardinal;
```

which is a single reservation statement that assigns types to four variables (`A`, `B`, `f`, and `M`) is replaced by four reservation statements, each of which assigns a type to a single variable:

```
reserve A for set;
reserve B for non empty set;
reserve f for Function of A, B;
reserve M for Cardinal;
```

When reserved variables are normalized in this way, one can eliminate some false positive dependencies. A theorem in which, say, the variable `f` occurs freely but which has nothing to do with cardinal numbers has the type `Function of A,B` in the presence of both the first and the

second sequences of reserved variables. If the first reservation statement is deleted, the theorem becomes ill-formed because \mathbf{f} no longer has a type. But the reservation statement itself directly requires that the type **Cardinal** of cardinal numbers is available, and thus indirectly requires a part of the development of cardinal numbers. If the theorem has nothing to do with cardinal numbers, this dependency is clearly specious.

These rewritings do not affect the semantics of the Mizar article.

Decomposition. For every normalized article A in the Mizar Mathematical Library, extract the sequence $\langle I_1^A, I_2^A, \dots, I_n^A \rangle$ of its toplevel items, each of which written to a “microarticle” A_k that contains only I_k^A and whose environment is that of A and contains each A_j ($j < k$).

Minimization. For every article A of the Mizar Mathematical Library and every microarticle A_n of A , do a brute-force minimization of smallest environment \mathcal{E}_{A_n} such that A_n is Mizar-verifiable.

The brute-force minimization works as follows. Given a microarticle A , we successively trim the environment^[10] For each item kind we have a sequence s of imported items $\langle a_1, \dots, a_n \rangle$, from which we find a minimal sublist s' with respect to which A is Mizar-verifiable^[11] Applying this approach for all Mizar item kinds, for all microarticles A_k , for all articles A of the MML is a rather expensive computation (for some Mizar articles, this process can take several hours). It is much slower than the method used for Coq described in Section 3. However the result is truly minimized, which is important for many applications of dependencies. Additionally, the minimization can be made significantly faster by applying heuristics, see Section 6.4 for a learning-assisted approach.

5 Comparison of the Methods

Some observations comparing the Coq and Mizar dependency computation can be drawn generally, without comparing the actual data as done in the following sections.

Dependencies in the case of CoRN are generated by hooking into the actual code and are thus quite exactly mirroring the work of the proof assistant. In the case of Mizar, dependencies are approximated from above. The dependency graph in this case starts with an over-approximation of what is known to be sufficient for an item to be Mizar-verifiable and then successively refines this over-approximation toward a minimal set of sufficient conditions. A significant difference is that the dependencies in Coq are not minimized. The dependency tracking there tells us exactly the dependencies that were used by Coq (in the particular context) when a certain command is run. For example, if the context is rich, and redundant dependencies are used by some exhaustive strategies, we will not detect their redundancy. On the other hand, in Mizar we do not rely

¹⁰ Namely, theorems, schemes, top-level lemmas, definitional theorems, definientia, patterns, registrations, and constructors. See [7] for a discussion of these item kinds.

¹¹ There is a (possibly non-unique) minimal sublist, since we assume that A is Mizar-verifiable to begin with.

on its exact operation. There, we exhaustively minimize the set of dependencies until an error occurs. This process is more computationally intensive. However, it does guarantee minimality (relative to the proof assistant’s power) which is interesting for many of the applications mentioned below.

Another difference is in the coverage of non-logical constructs. Practically every resource necessary for a verification of a Mizar article is an explicit part of the article’s environment. Thus, it is easy to minimize not just the strictly logical dependencies, but also the non-logical ones, like the sets of symbols and notations needed for a particular item, or particular automations like definitional expansions. For LCF-based proof assistants, this typically implies further work on the dependency tracking.

6 Evaluation, Experiments, and Applications

6.1 Dependency Extraction for CoRN and MML

We use the dependency extraction methods described in [3] and [4] to obtain fine dependency data for the CoRN library and an initial 100 article fragment of the MML. As described above, for CoRN, we use the dependency exporter implemented directly using the Coq code base. The export is thus approximately as fast as the Coq processing of CoRN itself, taking about 40 minutes on contemporary hardware. The product are for each CoRN file a corresponding file with dependencies, which have altogether about 65 MB. This information is then post-processed by scripts of off-the-shelf tools into the dependency graph discussed below.

For Mizar and MML we use the brute-force dependency extraction approach discussed above. This takes significantly longer than Mizar processing alone; a number of preprocessing and normalization steps that need to be done when splitting articles into micro-articles also decreases performance. For our data this now takes about one day for the initial 100 article fragment of the MML. The main share of this time being spent on minimizing the large numbers of items used implicitly by Mizar. Note that in this implementation we are initially more interested in achieving completeness and minimality rather than efficiency, and a number of available optimizations can reduce this time significantly.^[12]

In order to compare the benefits of having fine dependencies, we also compute for each library the *full file-based dependency* graph for all items. These graphs emulate the current dumb file-based treatment of dependencies in these libraries: each time an item is changed in some file, all items in the depending files have to be re-verified. The two kinds of graphs for both libraries are then compared in Table [1].

The graphs confirm our initial intuition that having the fine dependencies will significantly speed up partial recompilation of the large libraries, which is especially interesting in the CoRN and MML formal wikis that we develop.^[13] For example, the average number of items that need to be recompiled when a

¹² See Section [6.4].

¹³ <http://mws.cs.ru.nl/mwiki/>, <http://mws.cs.ru.nl/cwiki/>

random item is changed has dropped about seven times for CoRN, and about five times for Mizar. The medians for these numbers are even more interesting, increasing to fifteen for Mizar. The difference between MML and CoRN is also quite interesting, but it is hard to draw any conclusions. The corpora differ in their content and use different styles and techniques.

Table 1. Statistics of the item-based and file-based dependencies for CoRN and MML

	CoRN/item	CoRN/file	MML-100/item	MML-100/file
Items	9 462	9 462	9 553	9 553
Deps	175 407	2 214 396	704 513	21 082 287
TDEps	3 614 445	24 385 358	7 258 546	34 974 804
P(%)	8	54.5	15.9	76.7
ARL	382	2 577.2	759.8	3 661.1
MRL	12.5	1 183	155.5	2 377.5

Deps Number of dependency edges

TDEps Number of transitive dependency edges

P Probability that given two randomly chosen items, one depends (directly or indirectly) on the other, or vice versa.

ARL Average number of items recompiled if one item is changed.

MRL Median number of items recompiled if one item is changed.

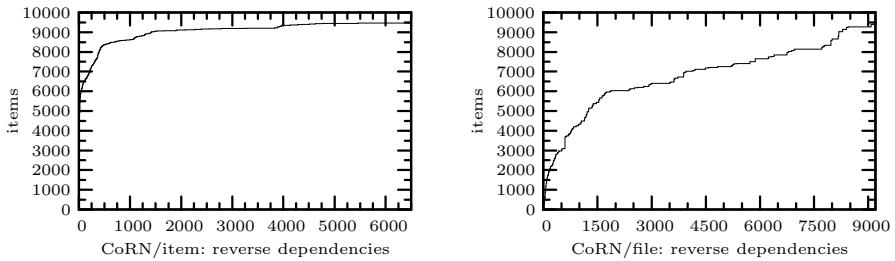


Fig. 1. Cumulative transitive reverse dependencies for CoRN: file-based vs. item-based

Table 2 shows statistics about the number and structure of *explicit* and *implicit* dependencies that we have done for Mizar. Explicit dependencies are anything that is already mentioned in the original text. Implicit dependencies are everything else, for example dependencies on type mechanisms (registrations, see also Section 6.4). Note that the ratio of implicit dependencies is very significant, which suggests that handling them precisely is essential for the learning and ATP experiments conducted in the next section.

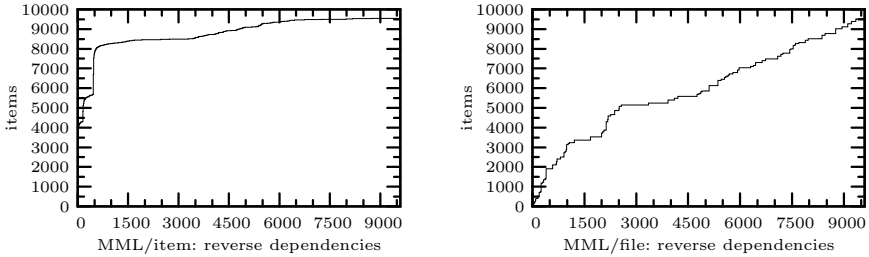


Fig. 2. Cumulative transitive reverse dependencies for MML: file-based vs. item-based

Table 2. Statistics of Mizar direct dependencies from and to different items

	theorem	top-level lemma	definition	scheme	registration
from	550 134	44 120	44 216	7 053	58 622
to	314 487	2 384	263 486	6 510	108 449

6.2 Dependency Analysis for AI-Based Proof Assistance

The knowledge of how a large number of theorems are proved is used by mathematicians to direct their new proof attempts and theory developments. In the same way, the precise formal proof knowledge that we now have can be used for directing formal automated theorem proving (ATP) systems and meta-systems over the large mathematical libraries. In [2] we provide an initial evaluation of the usefulness of our MML dependency data for machine learning of such proof guidance of first-order ATPs.

These experiments are conducted on a set of 2078 problems extracted from the Mizar library and translated to first-order ATP format. We emulate the growth of the Mizar library (limited to these 2078 problems) by considering all previous theorems and definitions when a new conjecture is attempted (i.e., when a new theorem is formulated by an author, requiring a proof). The ATP problems thus become very large, often containing thousands of the previously proved formulas as available axioms, which obviously makes automated theorem proving quite difficult [16, 12]. We run the state-of-the-art Vampire-SInE [9] ATP system on these large problems, and solve 548 of them (with a 10-second timelimit). Then, instead of attacking such large problems directly, we use machine learning to learn proof relevance from all previous fine-grained proof dependencies.

This technique works surprisingly well: in comparison with running Vampire-SInE directly on the large problems, the problems pruned by trained machine learners can be proved by Vampire in 788 resp. 824 cases, depending on the kind of machine learning method (naive Bayes, resp. kernel-based) applied. This means that the efficiency of the automated theorem proving is raised by 43.7% resp. 50.4% when we use the knowledge about previous proof dependencies. This is a very significant improvement in the world of automated theorem proving, where the search complexity is typically superexponential.

In [3] we further leverage this automated reasoning technique by scaling the dependency analysis to the whole MML, and attempting a fully automated proof for every MML theorem. This yields the so-far largest number of fully automated proofs over the whole MML, allowing us (using the precise formal dependencies of the ATP and MML proofs) to attempt an initial comparison of human and automated proofs in general mathematics.

6.3 Interactive Editing with Fine-Grained Dependencies

A particular practical use of fine dependencies (initially motivating the work done on Coq dependencies in [3]) is for advanced interactive editing. `tmEgg` [10] is a $\text{\TeX}_{\text{MACS}}$ -based user interface to Coq.¹⁴ Its main purpose is to integrate formal mathematics done in Coq in a more general document (such as course notes or journal article) without forcing the document to follow the structure of the formal mathematics contained therein.

For example, it does not require that the order in which the mathematical constructs appear in the document be the same as the order in which they are presented to Coq. As one would expect, the latter must respect the constraints inherent to the incremental construction of the formal mathematics, such as a lemma being proven before it is used in the proof of a theorem or a definition being made before the defined construct is used.

However, the presentation the author would like to put in the document may not strictly respect these constraints. For example, clarity of exposition may benefit from first presenting the proof of the main theorem, making it clear how each lemma being used is useful, and then only go through all lemmas. Or a didactic presentation of a subject may first want to go through some examples before presenting the full definitions for the concepts being manipulated.

`tmEgg` thus allows the mathematical constructs to be in any order in the document, and uses the dependency information to dynamically — and lazily — load any construct necessary to perform the requested action. For example, if the requested action is “check the proof of this theorem”, it will automatically load all definitions and lemmas used by the statement or proof of the theorem.

An interactive editor presents slightly different requirements than the batch recompilation scenario of a mathematical library described in Section 6.1. One difference is that an interactive editor needs dependency information, as part of the interactive session, for partial in-progress proofs. Indeed, if any in-progress proof depends on an item T , and the user wishes to change or unload (remove from the environment) T , then the part of the in-progress proof that depends on T has to be undone, even if the dependency is opaque.

¹⁴ The dependency tracking for Coq was actually started by the second author as part of the development of `tmEgg`. This facility has been already integrated in the official release of Coq. Since then this facility was extended to be able to treat the whole of the CoRN library. These changes are not yet included in the official release of Coq.

6.4 Learning Dependencies

In Section 6.2 it was shown that knowing the exact dependencies of previous proofs and learning from them can very significantly improve the chance of proving the next theorem fully automatically. Such AI techniques are however not limited just to fully automated proving, but can be used to improve the performance of many other tasks. One of them is also the actual process of obtaining exact minimal dependencies, as done for Mizar in Section 4.

This may seem impossible (or cyclic) at first: how can the process of determining minimal dependencies be practically improved by having minimal dependencies? The answer is that the (already computed) dependencies of the *previous* items can help significantly to guess the dependencies of the *next* item, and thus speed up the computation.

More precisely, for each Mizar item, we use its symbols (constructors) and explicit theorem references as the characterization (input features) for the learning. The output features (labels) are the registrations (implicit type mechanisms) needed for the item. For each item, we train the learner on all previous examples, and produce a prediction (registrations re-ordered by their predicted relevance) for the current item. The minimization algorithm is then modified to first do a binary search of this re-ordered list for its minimal initial segment sufficient for verifying the item, and then minimize this segment using the standard method. This is compared to the standard method on a random sample of 1000 theorems.

Tables 3 and 4 present our findings. The recommendation-assisted minimization algorithm was more than two times faster than unassisted minimization. Table 4 shows the speedup factors as the recommendation-assisted algorithm was applied to articles that are “deeper” in the MML, i.e., farther away from the set-theoretic axioms of Mizar. Our data confirms our intuition that, as the MML develops from the axioms of set theory to advanced mathematics, through smart recommendation we can considerably speed up dependency calculation.

Table 3. Summary of recommendation-assisted registration minimization

	Minimization time (sec)	Avg. minimization time/theorem (sec)
Unassisted	5755	5.76
Assisted	2433	2.43
Speedup Factor	2.37	
Number of cases		
Faster theorems	826	
Slower theorems	174	

7 Related Work

Related work exists in the first-order ATP field, where a number of systems can today output the axioms needed for a particular proof. Purely semantic

Table 4. Speedup through the library

Number of theorems	Unassisted time (sec)	Assisted time (sec)	Speedup Factor
200	143.81	90.18	1.59
400	967.90	541.70	1.79
600	1525.20	890.29	1.71
800	2828.82	1436.05	1.97
1000	5755.43	2433.26	2.37

(proof object) dependencies have been extracted several times for several ITPs, for example by Bertot and the Helm project for Coq [5,13,4], and Obua and McLaughlin for HOL Light and Isabelle. The focus of the latter two dependency extractions is on cross-verification, and are based on quite low-level (proof object) mechanisms. A higher-level¹⁵ semantic dependency exporter for HOL Light was recently implemented by Adams [1] for his work on HOL Light re-verification in HOL Zero. This could be usable as a basis for extending our applications to the core HOL Light library and the related large Flyspeck library. The Coq/CoRN approach quite likely easily scales to other large Coq libraries, like for example the one developed in the Math Components project [6]. Our focus in this work is wider than the semantic-only efforts: We attempt to get the full information about all implicit mechanisms (including syntactic mechanisms), and we are interested in using the information for smart re-compilation, which requires to track much more than just the purely semantic or low-level information.

8 Conclusion and Future Work

In this paper we have tried to show the importance and attractiveness of formal dependencies. We have implemented and used two very different techniques to elicit fine-grained proof dependencies for two very different proof assistants and two very different large formal mathematical libraries. This provides enough confidence that our approaches will scale to other important libraries and assistants, and our techniques and the derived benefits will be usable in other contexts.

Mathematics is being increasingly encoded in a computer-understandable (formal) and in-principle-verifiable way. The results are increasingly large inter-dependent computer-understandable libraries of mathematical knowledge. (Collaborative) development and refactoring of such large libraries requires advanced computer support, providing fast computation and analysis of dependencies, and fast re-verification methods based on the dependency information. As such automated assistance tools reach greater and greater reasoning power, the cost/benefit ratio of doing formal mathematics decreases.

Given our work on various parts of this program, providing exact dependency analysis and linking it to the other important tools seems to be a straightforward

¹⁵ By *higher-level* we mean tracking *higher-level* constructs, like use of theorems and tactics, not just tracking of the low-level primitive steps done in the proof-assistant's kernel.

choice. Even though the links to proof automation, fast large-scale refactoring, proof analysis, etc., are fresh, we believe that the significant performance boosts we've seen already sufficiently demonstrate the importance of good formal dependency analysis for formal mathematics, and for future mathematics in general.

References

1. Adams, M.: Introducing HOL Zero. In: Fukuda, K., Hoeven, J., Joswig, M., Takayama, N. (eds.) ICMS 2010. LNCS, vol. 6327, pp. 142–143. Springer, Heidelberg (2010)
2. Alama, J., Heskes, T., Kühlwein, D., Tsvitshivadze, E., Urban, J.: Premise selection for mathematics by corpus analysis and kernel methods. CoRR abs/1108.3446 (2012), <http://arxiv.org/abs/1108.3446>
3. Alama, J., Kühlwein, D., Urban, J.: Automated and Human Proofs in General Mathematics: An Initial Comparison. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18 2012. LNCS, vol. 7180, pp. 37–45. Springer, Heidelberg (2012)
4. Asperti, A., Padovani, L., Coen, C.S., Guidi, F., Schena, I.: Mathematical knowledge management in HELM. *Ann. Math. Artif. Intell.* 38(1-3), 27–46 (2003)
5. Bertot, Y., Pons, O., Pottier, L.: Dependency graphs for interactive theorem provers. Tech. rep., INRIA, report RR-4052 (2000)
6. Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: Packaging Mathematical Structures. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 327–342. Springer, Heidelberg (2009)
7. Grabowski, A., Kornilowicz, A., Naumowicz, A.: mizar in a nutshell. *Journal of Formalized Reasoning* 3(2), 153–245 (2010)
8. Haftmann, F., Wenzel, M.: Constructive Type Classes in Isabelle. In: Altenkirch, T., McBride, C. (eds.) TYPES 2006. LNCS, vol. 4502, pp. 160–174. Springer, Heidelberg (2007)
9. Hoder, K., Voronkov, A.: Sine Qua Non for Large Theory Reasoning. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 299–314. Springer, Heidelberg (2011)
10. Mamane, L., Geuvers, H.: A document-oriented Coq plugin for TeXmacs. In: Kauers, M., Kerber, M., Miner, R., Windsteiger, W. (eds.) MKM 2007 - Work in Progress. RISC Report, vol. 07-06, pp. 47–60. University of Linz, Austria (2007)
11. Matuszewski, R., Rudnicki, P.: Mizar: the first 30 years. *Mechanized Mathematics and Its Applications* 4, 3–24 (2005)
12. Meng, J., Paulson, L.C.: Lightweight relevance filtering for machine-generated resolution problems. *J. Applied Logic* 7(1), 41–57 (2009)
13. Pons, O., Bertot, Y., Rideau, L.: Notions of dependency in proof assistants. In: UITP 1998. Eindhoven University of Technology (1998)
14. Spitters, B., van der Weegen, E.: Type classes for mathematics in type theory. *Mathematical Structures in Computer Science* 21(4), 795–825 (2011)
15. Urban, J.: MoMM—fast interreduction and retrieval in large libraries of formalized mathematics. *International Journal on Artificial Intelligence Tools* 15(1), 109–130 (2006)
16. Urban, J., Hoder, K., Voronkov, A.: Evaluation of Automated Theorem Proving on the Mizar Mathematical Library. In: Fukuda, K., van der Hoeven, J., Joswig, M., Takayama, N. (eds.) ICMS 2010. LNCS, vol. 6327, pp. 155–166. Springer, Heidelberg (2010)
17. Wiedijk, F.: Mizar's Soft Type System. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 383–399. Springer, Heidelberg (2007)

Proof, Message and Certificate

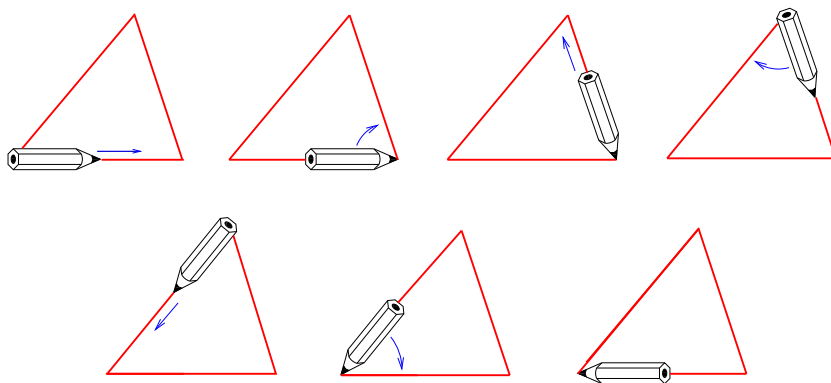
Andrea Asperti

Department of Computer Science
University of Bologna
asperti@cs.unibo.it

Abstract. The recent achievements obtained by means of Interactive Theorem Provers in the automatic verification of complex mathematical results have reopened an old and interesting debate about the essence and purpose of proofs, emphasizing the dichotomy between message and certificate. We claim that it is important to prevent the divorce between these two epistemological functions, discussing the implications for the field of mathematical knowledge management.

1 Introduction

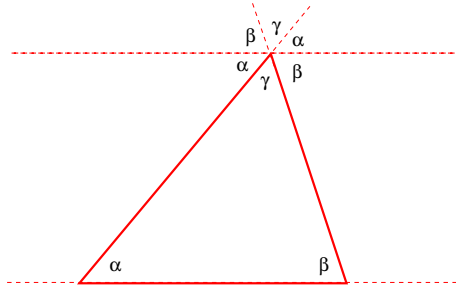
In December 2010, Aaron Sloman posted a message on the MKM mailing list that raised an interesting debate. His message was centered around the following “proof” of Euclid’s Theorem, stating that the internal angles of a triangle add up to a straight line (the argument was attributed to Mary Pardoe, a former student of Aaron Sloman). The proof just involves rotating a pencil through each of the angles at the corners of the triangle in turn, which results with the pencil ending up in its initial location but pointing in the opposite direction.



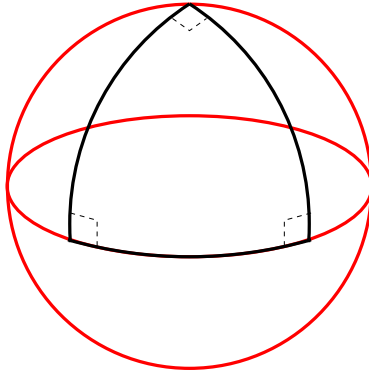
Sloman’s point was to show the relevance of graphical methods, in contrast with a logicistic approach, in the spirit of the book *Proofs without words: Exercises in Visual Thinking* by R. B. Nelsen [37] (see also D. Scott introduction to [42], or Jamnik’s book on diagrammatic proofs [28]). As Sloman expected, his post immediately raised a fierce debate in the community about the “validity”

of the above argument, with prestigious supporters on both sides. Dana Scott writes:

The proof is fine and really is the same as the classical proof. To see this, translate (by parallel translation) all the three angles of the triangle up to the line through the top vertex of the triangle parallel to the lower side. [...] Preservation of angles by parallel translation is justified by the Parallel Postulate.



In fact, the delicate point of the “proof” is the connection between three rotations performed at different positions in the space, and their translation to a same point, in order to sum them. This becomes evident if, instead of working on the plane, we repeat the pencil experiment on a sphere:



Of course, the problems related to the fifth-postulate and non-Euclidean geometries were evident to all people involved in the discussion: in fact the discussion rapidly switched from the *validity* of the proof to its *pedagogical value*. The supporters of the “proof” appreciated its nature of “thought experiment” (in Lakatos sense) not eventually leading to the expected result.¹ The fact that it fails on the sphere is actually informative, and can be used to better explain the relevance of the parallel postulate, that could otherwise be misunderstood. On the other side, detractors of the proofs were more concerned with the risk

¹ In Lakatos’ words [32], “after Columbus one should not be surprised if one does not solve the problem one has set out to solve.”

to present to students as a “valid” proof, an argument that is actually flawed. Quoting Arnon Avron:

If this “proof” is taught to students as a full, valid proof, then I do not see how the teacher will be able to explain to those students where the hell Euclid’s fifth postulate (or the parallels axiom) is used here, or even what is the connections between the theorem and parallel lines.

2 Message and Certificate

It is usually acknowledged (see e.g. [36]) that proofs have a double epistemological function, playing both the role of *message* and *certification*. In the first incarnation, the emphasis is entirely on *communication*: not only the proof is supposed to explain – by providing intuitions – the reasons for believing in the validity of a given statement, but it should also convey information about the line of thought used to conjecture the result and the techniques used for approaching it. In the second incarnation, the proof is supposed to provide a precise line of reasoning that can be verified in an objective and essentially mechanical way: you can follow and check the validity of the argument even without having a clear understanding of its *meaning*.

The debate about the actual role of proofs in mathematics (see also [30] for a recent survey) essentially concerns the different relevance attributed to the role of message or certificate.

A very common position among mathematicians is to firmly negate any deductive validity to proofs; G. H. Hardy, who is traditionally credited with reforming British mathematics by bringing rigor into it, described the notion of mathematical proof *as we working mathematicians are familiar with* in the following terms [22]:

There is strictly speaking no such thing as a mathematical proof; we can, in the last analysis, do nothing but point; [. . .] proofs are what Littlewood and I call gas, rhetorical flourishes designed to affect psychology, pictures on the board in the lecture, devices to stimulate the imagination of pupils.

The opposite position consists in negating any possibility of *communication* without a clear, objective and verifiable assessment of its actual content. The position is nicely summarized by the following words of de Bruijn² [16]

If you can’t explain your mathematics to a machine it is an illusion to think you can explain it to a student.

A simple example can probably help to understand the issue. Consider the problem of proving that the sum of the first n positive integers is equal to $\frac{n \cdot (n+1)}{2}$.

² See [3] for a deeper discussion of de Bruijn’s sentence.

A simple approach (anecdotally attributed to the precocious genius of Gauss³), is to write the sum horizontally forwards and backwards, observe that the sum of each column amounts to $n + 1$, and we have n of them, giving a total of $\frac{n \cdot (n+1)}{2}$.

– **Message**

$$\begin{array}{cccccc} 1 & 2 & \dots & n-1 & n & \\ n & n-1 & \dots & 2 & 1 & \\ \hline n+1 & n+1 & \dots & n+1 & n+1 & \end{array}$$

A seemingly simple proof can be given by induction: the case “ $n = 1$ ” is obvious, and the inductive case amounts to a trivial computation:

– **Certification**

$$\frac{(n-1) \cdot n}{2} + n = \frac{n \cdot (n+1)}{2}$$

The actual information provided by the two proofs is of course very different: Gauss’ “trick” gives us a general *methodology* suitable to be used not only in the given situation but, *mutatis mutandi*, in a wide range of similar problems; in contrast, the inductive proof is quite sterile and uninformative: if we do not know in advance the closed form of the progression (in general, the property we are aiming to), induction provides no hint to guess it. The interesting part of Gauss’ proof is its *message*, while the inductive proof is, in this case, a mere *certificate*. The importance of the message often transcends the actual relevance of the statement itself: the fact that the sum of the arithmetic progression is equal to $n(n+1)/2$ is of marginal interest, but Gauss’ technique is a major source of inspiration. This is why we are interested in proofs: because they embody the techniques of mathematics and shape the actual organization of this discipline into a structured collection of interconnected notions and theories. What we expect to gain from a solution of the $P \stackrel{?}{=} NP$ problem is not quite the knowledge about the validity of this statement, but a new insight into notions and techniques that appear to lie beyond the current horizon of mathematics. And this is also the main reason why we are interested in *formal* proofs: because the process of formalization obliges to a deeper, philosophical and scientific reflection of the logical and linguistic mechanisms governing the deployment and the organization of mathematical knowledge [3].

³ Brian Hayes, [25] collected several hundred accounts of the story of Gauss’s boyhood discovery of the “trick” for summing an arithmetic progression, comprising the following:

When Gauss was 6, his schoolmaster, who wanted some peace and quiet, asked the class to add up the numbers 1 to 100. “Class,” he said, coughing slightly, “I’m going to ask you to perform a prodigious feat of arithmetic. I’d like you all to add up all the numbers from 1 to 100, without making any errors.” “You!” he shouted, pointing at little Gauss, “How would you estimate your chances of succeeding at this task?” “Fifty-fifty, sir,” stammered little Gauss, “no more ...”

It is interesting to observe that Gauss' argument is not so easy to formalize, requiring several small properties of finite summations⁴. In particular, it relies on the following facts: (perm) the sum is unchanged under permutation of the addends, (distr) $\sum_{i=1}^n a_i + \sum_{i=1}^n b_i = \sum_{i=1}^n (a_i + b_i)$ and (const) summing n constant elements c yields $n \cdot c$. The best approximation we can do of a "formal" version of Gauss' argument looks something like the following

$$\begin{aligned} 2 \cdot \sum_{i=1}^n i &= \sum_{i=1}^n i + \sum_{i=1}^n i \\ &= \sum_{i=1}^n i + \sum_{i=1}^n (n - i + 1) \\ &= \sum_{i=1}^n (i + n - i + 1) = \\ &= \sum_{i=1}^n (n + 1) \\ &= n \cdot (n + 1) \end{aligned}$$

One could easily wonder if, in this process, the original *message* has not been entirely lost⁵. What is particularly annoying is that, in the formal proof, there isn't a *single* crucial step that embodies the essence of the proof: it is a clever combination of perm, distr and const that makes it work. In fact, the nice point of the graphical representation is to put them together in a single picture: adding rows is the same as adding columns, and each column sum up to the same value. But this raises another interesting issue: namely, if what makes Gauss' argument so appealing is not due to an intrinsic property of the proof but to the fact that it suits particularly well to the intellectual (and sensorial, synoptical) capacities of the human mind.

Most of the proofs in elementary arithmetic have a similar nature. For instance, this is a typical proof⁶ of the main property of the Euler ϕ function, computing for any positive integer n the number of integers between 1 and n relative prime to n .

Proposition. $\sum_{d|n} \phi(d) = n$.

Proof. Consider the n rational numbers $\frac{1}{n}, \frac{2}{n}, \frac{3}{n}, \dots, \frac{n-1}{n}, \frac{n}{n}$. Reduce each to lowest terms; i.e., express each number as quotient of relative prime integers. The denominators will all be divisors of n . If $d|n$, exactly $\phi(d)$ of our numbers will have d in the denominator after reducing to lowest terms. Thus $\sum_{d|n} \phi(d) = n$.

Again, a formal proof⁷ requires a not trivial play with summations properties that is eventually going to hide the intuitive argument so readily communicated by the previous sketch (this is also why a good library of "big ops"⁸ is *essential* for any formal development involving combinatorics).

In general, we should accept the fact that there will be many proofs that, once formalized, will loose that degree of *unexpectedness, combined with inevitability and economy* that according to Hardy⁹ make the beauty of a mathematical

⁴ Since summation is defined by recursion, most proofs of its properties require recursion too.

⁵ On the other side, one could also wonder if, after all, Gauss's argument doesn't hide too many details that are worth to be spelled out.

proof. But, mathematics itself is entering a new era of results requiring *extraordinarily long and difficult megaproofs, sometimes relying heavily on computer calculations, and leaving a miasma of doubt behind them* [35]. Maybe, enumeration by cases, *one of the duller forms of mathematical argument* in Hardy's opinion [23], could turn out to be the only viable way to achieve a result, as in the case of the four color theorem [21].

According to Lakatos, *simplicity was the eighteenth-century idea of mathematical rigor* [32], and maybe as we already observed in [5] we should just learn to appreciate a different, and less archaic, kind of beauty.

3 A Social Process

Strictly intertwined with the dichotomy between message and certificate is the discussion about the actual nature of the process aimed to assess the validity of a mathematical argument: ⁶ a social process, or an objective, almost mechanical activity (see [5] for a recent survey on this topic). The two positions can be summarized by the following quotations:

social/subjective

We believe that, in the end, it is a social process that determines whether mathematicians feel confident about a theorem.

– R. A. De Millo, R. J. Lipton, A. J. Perlis [17]

decidable/objective

A theorem either can or cannot be derived from a set of axioms. I don't believe that the correctness of a theorem is to be decided by a general election.

– L. Lamport [33]

The main argument usually alleged by the paladins of the “social” perspective, is the practical impossibility of developing fully formal demonstrations, due to the “nearly inconceivable” length of a deduction from first principles.

Russell did succeed in showing that ordinary working proofs can be reduced to formal, symbolic deductions. But he failed, in three enormous, taxing volumes, to get beyond the elementary facts of arithmetic. He showed what can be done in principle and what cannot be done in practice.

[...] A formal demonstration of one of Ramanujan's conjectures assuming set theory and elementary analysis would take about two thousand pages.

– R. A. De Millo, R. J. Lipton, A. J. Perlis [5]

Even Bourbaki⁷ who is traditionally enlisted in the ranks of the formalist school [34], labels the project of formalizing mathematics as *absolutely unrealizable*

⁶ It is important to stress that the debate is not about the overall mathematical activity, that is indubitably a social process (at least, in the same complex, and often conflictual way, artistic creation is), but is really confined to correctness checking.

⁷ Bourbaki did not particularly like logic, that he considered as a mere tool: “logic, as far as we mathematicians are concerned, is no more and no less than the grammar of the language which we use, a language which had to exist before the grammar could be constructed” [14].

The tiniest proof at the beginning of the Theory of Sets would already require several hundreds of signs for its complete formalization.

– Bourbaki [13]

As we observed in [5], the argument is reminiscent of the general disbelief about the possibilities of writing long programs at the beginning of the fifties: in fact, they were reasoning in terms of assembly languages, and the mistake was due to the inability to conceive a process of automatic translation from a high level programming language to a machine-understandable code. The same is true for formal proofs: a modern interactive prover is precisely a tool interpreting a high level mathematical language (we shall discuss them in the next section) into a set of low level logical instructions automatically checked for correctness by the machine.

The arrival of the computer changes the situation dramatically. [...] checking conformance to formal rules is one of the things computers are very good at. [...] the Bourbaki claim that the transition to a completely formal text is routine seems almost an open invitation to give the task to computers.

– J. Harrison [24]

It is important to observe that the high level “proof” can be *arbitrarily* compact: it only depends on how much time you are ready to pay for the translation. From this point of view, a “proof” is *any information* sufficient to make decidable the correctness of a statement (that, in principle, fixed the formal system, is only semidecidable). For instance, an upper bound to its dimension (or any function of it) is a perfectly formal (and decidedly compact) proof. So, something like “10 lines” should be accepted as a perfectly formal argument (that we shall henceforth call “à la Fermat”, paying homage to to his actual inventor): we have just to generate and check all proof-terms of the formal language within the given bound: if one of them proves the statement the argument is correct, and otherwise it is wrong.⁸

This sheds new light on the dichotomy between message and certificate: in fact, what kind of message can be found in a proof *à la Fermat*? In other words, it is true that formal proofs can be arbitrarily compact, but it is equally true that the certificate can be *arbitrarily distant* from *any* message.

The divorce between the notions of *message* and *certification* induced by computer proof assistants has been already remarked by Dana Mackenzie in a recent article that appeared on Science, where he apparently attributes a positive value to such a separation:

⁸ In a recent invited talk at CICM 2011, Trybulec suggested to use time complexity to make a distinction between *proofs* and *traces*, reserving the title of proofs only for those certificates that can be checked with a “reasonable” (say, polynomial) complexity. For Automatic Interactive Provers it is of course important to be able to perform proof checking in a reasonable amount of time (hence, certificates are usually proofs in Trybulec sense). At the current state of the art, the dimension of formal certificates is sensibly more verbose (2 to 10 times larger) than the corresponding high level mathematical proof (see [4] for a discussion).

Ever since Euclid, mathematical proofs have served a dual purpose: certifying that a statement is true, and explaining why it is true. Now those two epistemological functions may be divorced. In the future, the computer assistant may take care of the certification and leave the mathematician to look for an explanation that humans can understand.

– D. Mackenzie [35]

Mackenzie’s argument, however, is pretty weak: if the certificate is divorced from the message, it is enough (up to the adequacy of the encoding) to certify the correctness of the statement, but it says nothing about the correctness of its supposed “explanation”. Often, what is doubtful is not the validity of statements, but of their proofs: so, if message and certificate are distant, we are essentially back to the original situation: we are not sure if the “message” the author is trying to communicate to the reader is correct.

The usual objections raised by mathematicians to the issue of automatic verification of statements concern either the correctness of the proof checker itself or the correctness of the encoding of the problem (*adequacy*). For the first point, proof checkers are single applications, often open source, in competition with each other and subject to a severe experimental verification (see also [19] for a list of properties that could strengthen our conviction that a proof checker is reliable). For the second point, sometimes underestimated in the proof assistant community, we should observe that the adequacy of the formalization only depends on the formulation of definitions and statements, and checking that they reflect their intended meaning is a much easier task than checking the correctness of the entire proof. For instance, Gonthier emphasizes that the formal statement of the 4-color theorem [21], including “all definitions” required to understand it, fits on one A4 page; while we believe that this is somewhat an overstatement and more pages are actually required, the point is that, if you trust the proof assistant, you just need to understand and verify a small amount of information.

A more substantial objection concerns the real added value provided by having a formal proof *in case we are not able to convey a human readable message out of it*. We would be in the odd situation to know the existence of a proof, but to have a pretty vague idea of how it concretely works.⁹

⁹ With have a similar situation with *proofs by reflection* [15]. The basic idea of this technique is that checking a proof involves running some certified decision procedure. For instance, in order to compare two regular expressions, we can build the corresponding automata and run a suitable bisimulation algorithm. In this case too, we have no direct grasp of the specific proof, but the big difference is that we have a clear understanding of the reasons why the proof is correct, i.e. of the metatheory underlying the approach; if the algorithm is implemented correctly (and this can be mechanically verified), then we can be confident in the proof.

4 Declarative vs. Procedural

It is usually believed that declarative languages are in a better position than procedural ones to preserve the relation between message and certificate in the realm of formal mathematics.

Before entering in this discussion, we would like to attempt a clarification of the two classes of languages. To this end, we make a simple analogy with chess. A chess game can be described in essentially two ways: as a sequence of *moves* or as a sequence of *positions* (see Figure [1](#)). In the first case, *positions are implicit*: they can be reconstructed by executing a subsequence of the moves; in the second case, *moves are implicit*: they can be deduced by the difference between consecutive positions. Moves and positions are simple examples of, respectively, a procedural and a declarative language.

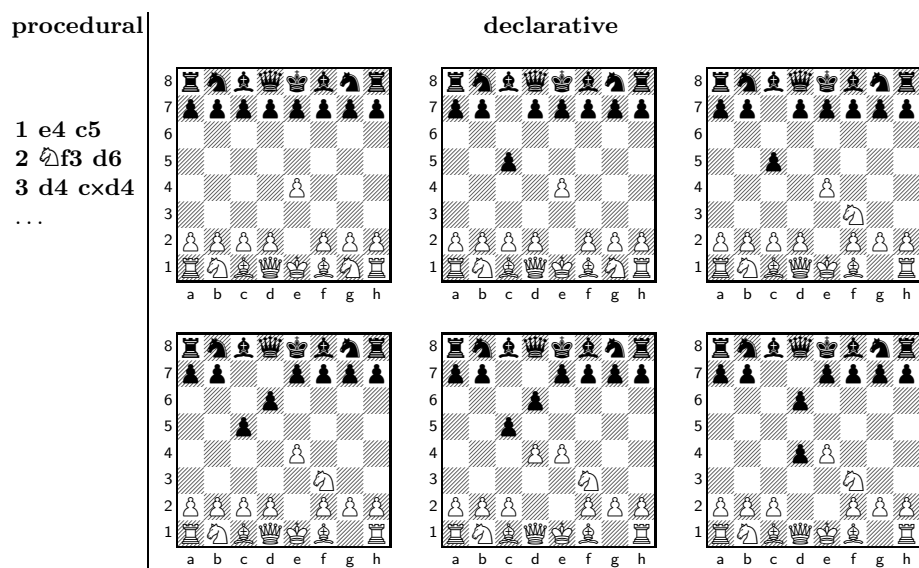


Fig. 1. Moves and Positions

In the case of logic, you have a similar situation. A proof of P_n , in Hilbert's acceptance, is a sequence of formulae P_1, \dots, P_n where each P_i is either an axiom or is obtained from formulae preceding P_i in the sequence by means of suitable (fixed) logical rules. This description is actually redundant: you may just give the sequence of rules (procedural description) leaving implicit the sequence of formulae, or you may give the whole sequence of intermediate results P_1, \dots, P_n leaving to the reader the (easy) task to understand the logical rule required for each inference (declarative description).

The relative merits of the two representations should be clear. A procedural description is *very compact* but quite unreadable: the point is that each move refers to a state implicitly defined by the previous steps. To understand the

proof, you should be able to figure out in your mind the state of the system after each move. In the case of chess, this is still possible for a trained human, since the board is a relatively simple structure, moves are elementary operations, and games are not too long. However, it becomes practically impossible as soon as you deal with more complex situations, such as symbolic logic.¹⁰

On the other side, declarative descriptions provide, at each instant, a full description of the current state: since the evolution does not depend on the past (the game is history free), you do not need to remember or rebuild any information and may entirely focus on the given state. Declarative descriptions are hence immediately readable¹¹ but also (as it is evident in the case of chess), much more *verbose*.

The gap between procedural and declarative languages is not so large as it may appear at first sight: in fact, they *complement* each other and *integrate together* very well. For instance, when discussing a chess game in the procedural style it is customary to explicitly draw the state of the board at particularly interesting or instructive places (see Figure 2 - Fisher vs Larsen, Portoroz 1958, Sicilian Defense, Yugoslavian Attack at the Dragon Variation).

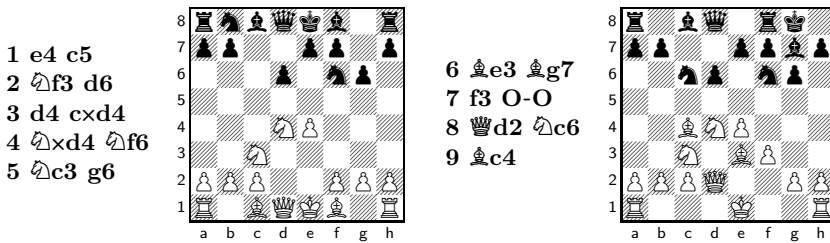


Fig. 2. Cuts

In the terminology of logic, this operation is called a *cut*, since it divides a complex description (a game, a proof) into smaller components, each one with an independent interest (not every position of the board is worth a draw, in the same way as not every intermediate logical step is worth a cut). The tendency, among interactive provers adopting a procedural style, to promote the use of cuts and a more structured description of the proof (and hence to implicitly move towards a more declarative and readable style of proofs) is clearly testified by the most recent applications such as Ssreflect [20] or Matita [38,8].

¹⁰ The criticism that procedural languages lack readability is a bit unfair: the point is that they *are not meant* to be read, but to be interactively re-executed.

¹¹ By “readability” we mean here the mere possibility to follow more easily the chain of reasoning in a proof; of course this does not imply a real grasp of the information it is supposed to convey. For instance, it is difficult to learn chess by just studying past games, no matter how they are represented, without auxiliary expertise.

On the other hand, we can make the declarative description less verbose by simply augmenting the granularity of steps. For instance, without any loss of information, we can decide to represent the board at each player-opponent move instead of considering single half moves (see Figure 3). But we could arbitrarily

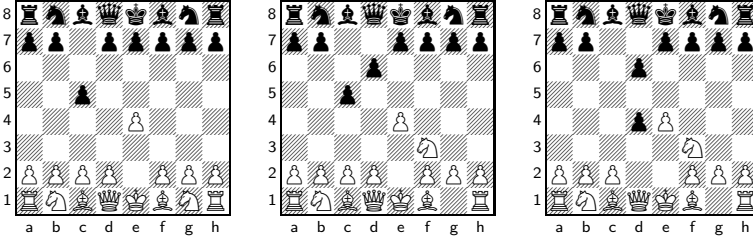


Fig. 3. Augmenting the granularity

decide to represent the board every, say, pair of moves, *relying on the reader's ability to fill in the missing information*. The granularity could of course depend on the state of the game, and the knowledge we expect from the reader. For instance, chess *openings* have been deeply investigated, and each good chess player immediately recognizes a particular opening from the state of the board in which it ends up: so, a procedural part of several moves can be left implicit and we can immediately jump in the middle of the game. In the case of declarative languages for interactive provers, the intended reader is the machine, that is, a device with limited intelligence: as a consequence the granularity cannot be too coarse-grained.¹² However, we can further reduce it by providing suitable proof hints to the machine, or explicit procedural fragments.

Having clarified that procedural and declarative languages are not mutually exclusive, and may easily integrate (see also [11,41]), there is however an important observations that must be made.

In declarative languages, the level of granularity is somehow *imposed by the intelligence of the machine*, and there is no reason to believe this corresponds with the requirement of a good exposition in human terms. As we already observed, granularity can be modified by adding suitable proof hints, but again these proof-hints, similarly to procedural fragments, must be machine understandable, and they may hardly improve the readability of the text (they are essentially meant to reduce verbosity).

On the other side, a procedural proof can be more or less arbitrarily split in smaller fragments, for expository purposes. Moreover, the kind of enrichment we

¹² This use of the machine, meant to relieve the user from the burden to fill in relatively trivial steps by automatically completing the missing gaps (the underlying “logical glue” of the mathematical language) is called *small scale automation* in [9], in contrast with *large scale automation* referring to the more creative and complex activity to help the user in the process of figuring out the actual proof. The two kinds of automation seem to have different requirements and can possibly deserve different approaches and solutions.

can do on the text can be entirely aimed for humans: if we ask an automatic chess player to print the state of the board at a given instant we do not necessarily expect the player to be able to reparse this representation, purely meant for human convenience (even if we may agree that it could be a desirable property). In other terms, the kind of enrichment corresponding with a “cut” does not necessarily need to be a *formal statement*: it could be a comment, a picture, a diagram, an animation, or whatever. The same is essentially true with a declarative language, but in this case any additional comment or explanation is eventually going to interfere with the original vernacular, adding a confusing level of “explanation” to a language that was already supposed to be self-explanatory.

5 A Complex Problem

The complexity of the problem of preserving the relation between message and certificate can be understood by an analogy¹³ with software, where we have essentially the same situation: writing a program requires understanding and solving a problem, but it is extremely difficult to extract such a knowledge (the *message*) from the final code (playing the role of *certification*). The major investment, in programming as well as in formalization, is not the actual writing up of the program, but the preliminary phase of analysis, planning and design; it is a real pity that this information gets essentially lost in the resulting encoding. In spite of the evident relevance of the problem, we have assisted during the last decades to the substantial failure of many interesting projects aimed to improve writing and readability of programs. A relevant example is *literate programming* [31], that in the intention of Knuth was not just a way to produce high-quality formatted documentation, but a methodology aimed to improve the quality of the software itself, forcing programmers to explicitly state the relevant concepts behind their code. Literate programming was meant to represent what Knuth called the “web of abstract concepts” hiding behind the design of software; in particular, it supported the possibility to change the order of the source code from a machine-imposed sequence to one more convenient to the human mind.

The reasons why literate programming failed to have a significant impact on software development are not so evident. We could probably admit that in modern programming languages (both object oriented and functional), the kind of abstraction mechanisms provided by the language are already sufficient to prevent that “dictatorship” of the machine that Knuth seemed to suffer in a particular way.¹⁴ This is probably enough to explain why, at present, a simple

¹³ The analogy we are making here, comparing programs with proofs, is essentially the so-called Curry-Howard analogy [26]. The fact of finding in the two realms the same dichotomy message/certification proves once again the deep philosophical implications of this correspondance that largely transcend the technical aspects of proof theory.

¹⁴ In any case, it is surely more interesting, and probably more useful, to improve the programming language, than solving the problem at a meta level, via a generative approach.

documentation generator like say, Doxygen,¹⁵ is largely more popular than the more sophisticated and ambitious literate programming approach.

It is likely to expect a similar situation in the realm of interactive provers. Nowadays, procedural languages for interactive provers permit to adhere with sufficient precision to the natural “flows of thoughts”, and techniques and methodologies like small scale reflection [20], mathematical components [18] or small scale automation [9] are meant to further improve on this situation. So, a simple documentation generators is likely to be more rapidly adopted by users of interactive provers than a more sophisticated authoring interface.¹⁶

Many available proof assistant already provide functionalities for enriched HTML presentation of their libraries, like the Coqdoc tool for the Coq System. These tools allow to produce interesting ebooks, like for instance the “Software Foundations” course at <http://www.cis.upenn.edu/~bcpierce/sf/>. The next natural step would consist in improving on line interactivity with the document, especially for editing and execution. An additional layer (called Proviola) is currently being developed [39] on top of coqdoc, with the goal of providing more dynamic presentations of the Coq formalizations. Several system are developing web-interfaces for their application (see [29,7]), many of them inspired by wikis [40,11]. The wiki-approach looks particularly promising: a large repository of mathematical knowledge may be only conceived as a collaborative working space. This is especially true in a formal setting where, in the long term, re-use of mathematical components will be the crucial aspect underpinning the success of interactive theorem provers.

Acknowledgments. This paper is partially based on two talks given by the author, one at the Tata Institute of Technology, Mumbai (India) in 2009 and another at the Summer School of Logic of the Italian Association for Logic and Applications, held in Gargnano, Italy, in August 2011. I would like to thank, respectively, Raja Natarajan and Silvio Ghilardi for offering me these opportunities. I would also like to thank the anonymous reviewers for their detailed and stimulating comments.

References

1. Alama, J., Brink, K., Mamane, L., Urban, J.: Large Formal Wikis: Issues and Solutions. In: Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.) *Calculemus/MKM 2011*. LNCS, vol. 6824, pp. 133–148. Springer, Heidelberg (2011)
2. Asperti, A., Armentano, C.: A page in number theory. *Journal of Formalized Reasoning* 1, 1–23 (2008)
3. Asperti, A., Avigad, J.: Zen and the art of formalization. *Mathematical Structures in Computer Science* 21(4), 679–682 (2011)

¹⁵ <http://www.doxygen.org>

¹⁶ This does not imply that one is better than the other; in the long run, a highly sophisticated authoring environment can still be the better solution.

4. Asperti, A., Sacerdoti Coen, C.: Some Considerations on the Usability of Interactive Provers. In: Autexier, S., Calmet, J., Delahaye, D., Ion, P.D.F., Rideau, L., Rioboo, R., Sexton, A.P. (eds.) AISC 2010. LNCS, vol. 6167, pp. 147–156. Springer, Heidelberg (2010)
5. Asperti, A., Geuvers, H., Natarajan, R.: Social processes, program verification and all that. *Mathematical Structures in Computer Science* 19(5), 877–896 (2009)
6. Asperti, A., Ricciotti, W.: About the Formalization of Some Results by Chebyshev in Number Theory. In: Berardi, S., Damiani, F., de’Liguoro, U. (eds.) TYPES 2008. LNCS, vol. 5497, pp. 19–31. Springer, Heidelberg (2009)
7. Asperti, A., Ricciotti, W.: A Web Interface for Matita. In: Jeuring, J., et al. (eds.) CICM 2012. LNCS (LNAI), vol. 7362, pp. 417–421. Springer, Heidelberg (2012)
8. Asperti, A., Ricciotti, W., Sacerdoti Coen, C., Tassi, E.: The Matita Interactive Theorem Prover. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 64–69. Springer, Heidelberg (2011)
9. Asperti, A., Tassi, E.: Superposition as a logical glue. *EPTCS* 53, 1–15 (2011)
10. Avigad, J., Donnelly, K., Gray, D., Raff, P.: A formally verified proof of the prime number theorem. *ACM Trans. Comput. Log.* 9(1) (2007)
11. Barendregt, H.: Towards an interactive mathematical proof language. In: Kamareddine, F. (ed.) *Thirty Five Years of Automath*, pp. 25–36. Kluwer (2003)
12. Bertot, Y., Gonthier, G., Ould Biha, S., Pasca, I.: Canonical Big Operators. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 86–101. Springer, Heidelberg (2008)
13. Bourbaki, N.: The architecture of mathematics. *Monthly* 57, 221–232 (1950)
14. Bourbaki, N.: *Theory of Sets. Elements of mathematics*. Addison Wesley (1968)
15. Boutin, S.: Using Reflection to Build Efficient and Certified Decision Procedures. In: Ito, T., Abadi, M. (eds.) TACS 1997. LNCS, vol. 1281, pp. 515–529. Springer, Heidelberg (1997)
16. De Bruijn, N.G.: Memories of the automath project. Invited Lecture at the Mathematics Knowledge Management Symposium, November 25-29. Heriot-Watt University, Edinburgh (2003)
17. De Millo, R.A., Lipton, R.J., Perlis, A.J.: Social processes and proofs of theorems and programs. *Commun. ACM* 22(5), 271–280 (1979)
18. Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: Packaging Mathematical Structures. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 327–342. Springer, Heidelberg (2009)
19. Geuvers, H.: Proof Assistants: history, ideas and future. *Sadhana* 34(1), 3–25 (2009)
20. Gonthier, G., Mahboubi, A.: An introduction to small scale reflection in coq. *Journal of Formalized Reasoning* 3(2), 95–152 (2010)
21. Gonthier, G.: Formal proof – the four color theorem. *Notices of the American Mathematical Society* 55, 1382–1394 (2008)
22. Hardy, G.H.: Mathematical proof. *Mind* 38, 1–25 (1928)
23. Hardy, G.H.: *A Mathematician’s Apology*. Cambridge University Press, London (1940)
24. Harrison, J.: Formal proof – theory and practice. *Notices of the American Mathematical Society* 55, 1395–1406 (2008)
25. Hayes, B.: Gauss’s day of reckoning. *American Scientist* 4(3), 200–207 (2006)
26. Howard, W.A.: The formulae-as-types notion of construction. In: Seldin, J.P., Hindley, J.R. (eds.) *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pp. 479–490. Academic Press, Boston (1980)
27. Ireland, K., Rosen, M.: *A Classical Introduction to Modern Number Theory*. Springer (2006)

28. Jamnik, M.: *Mathematical Reasoning with Diagrams: from intuition to automation*. CSLI Press, Stanford (2001)
29. Kaliszyk, C.: Web interfaces for proof assistants. *Electr. Notes Theor. Comput. Sci.* 174(2), 49–61 (2007)
30. Kerber, M.: Proofs, Proofs, Proofs, and Proofs. In: Autexier, S., Calmet, J., Delahaye, D., Ion, P.D.F., Rideau, L., Rioboo, R., Sexton, A.P. (eds.) *AISC 2010*. LNCS, vol. 6167, pp. 345–354. Springer, Heidelberg (2010)
31. Knuth, D.E.: *Literate Programming*. Center for the Study of Language and Information (1992)
32. Lakatos, I.: *Proofs and Refutations: The Logic of Mathematical Discovery*. Cambridge University Press (1976)
33. Lamport, L.: Letter to the editor. *Communications of the ACM* 22, 624 (1979)
34. Lee, J.K.: Philosophical perspectives on proof in mathematics education. *Philosophy of Mathematics Education Journal* 16 (2002)
35. MacKenzie, D.: What in the name of Euclid is going on here? *Science* 207(5714), 1402–1403 (2005)
36. Mackenzie, D.: *Mechanizing Proof*. MIT Press (2001)
37. Nelsen, R.B.: *Proofs without Words: Exercises in Visual Thinking*. The Mathematical Association of America (1997)
38. Coen, C.S., Tassi, E., Zacchiroli, S.: Tincals: step by step tacticals. In: *Proceedings of User Interface for Theorem Provers 2006*. *Electronic Notes in Theoretical Computer Science*, vol. 174, pp. 125–142. Elsevier Science (2006)
39. Tankink, C., Geuvers, H., McKinna, J., Wiedijk, F.: Proviola: A Tool for Proof Re-animation. In: Autexier, S., Calmet, J., Delahaye, D., Ion, P.D.F., Rideau, L., Rioboo, R., Sexton, A.P. (eds.) *AISC 2010*. LNCS, vol. 6167, pp. 440–454. Springer, Heidelberg (2010)
40. Urban, J., Alama, J., Rudnicki, P., Geuvers, H.: A Wiki for Mizar: Motivation, Considerations, and Initial Prototype. In: Autexier, S., Calmet, J., Delahaye, D., Ion, P.D.F., Rideau, L., Rioboo, R., Sexton, A.P. (eds.) *AISC 2010*. LNCS, vol. 6167, pp. 455–469. Springer, Heidelberg (2010)
41. Wiedijk, F.: Formal proof sketches. In: Fokink, W., van de Pol, J. (eds.) *7th Dutch Proof Tools Day, Program + Proceedings*. CWI, Amsterdam (2003)
42. Wiedijk, F. (ed.): *The Seventeen Provers of the World*. LNCS (LNAI), vol. 3600. Springer, Heidelberg (2006)

Challenges and Experiences in Managing Large-Scale Proofs

Timothy Bourke¹, Matthias Daum^{1,2}, Gerwin Klein^{1,2}, and Rafal Kolanski^{1,2}

¹ NICTA, Sydney, Australia*

² The University of NSW, Sydney, Australia

{timothy.bourke,matthias.daum,gerwin.klein,rafal.kolanski}@nicta.com.au

Abstract. Large-scale verification projects pose particular challenges. Issues include proof exploration, efficiency of the edit-check cycle, and proof refactoring for documentation and maintainability. We draw on insights from two large-scale verification projects, L4.verified and Verisoft, that both used the Isabelle/HOL prover. We identify the main challenges in large-scale proofs, propose possible solutions, and discuss the *Levity* tool, which we developed to automatically move lemmas to appropriate theories, as an example of the kind of tool required by such proofs.

Keywords: Large-scale proof, Isabelle/HOL, Interactive theorem proving.

1 Introduction

Scale changes everything. Even simple code becomes hard to manage if there is enough of it. The same holds for mathematical proof—the four-colour theorem famously had a proof too large for human referees to check [3]. The theorem was later formalised and proved in the interactive proof assistant Coq [4] by Gonthier [7,8], removing any doubt about its truth. It took around 60,000 lines of Coq script. While an impressive result, this was not yet a large-scale proof in our sense. Verifications another order of magnitude larger pose new challenges.

To gain a sense of scale for proof developments, we analysed the Archive of Formal Proofs (AFP) [12], a place for authors to submit proof developments in the Isabelle/HOL proof assistant [14]. The vast majority of the over 100 archive entries are proofs that accompany a separate publication. Submissions contain from 3 to 3,938 lemmas per entry, from 145 to 80,917 lines of proof, and from 1 to 151 theory files. The average AFP entry has 340 lemmas shown in 6,000 lines of proof in 10 theory files. Fig. 1 shows the size distribution of entries ordered by submission date. The spikes between the majority of small entries are primarily PhD theses. Beyond the AFP, an average PhD thesis in Isabelle/HOL is about 30,000 lines of proof in our experience.

In recent years, the first proofs on a consistently larger scale have appeared, in particular the verification of an optimising compiler in the CompCert project [13],

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

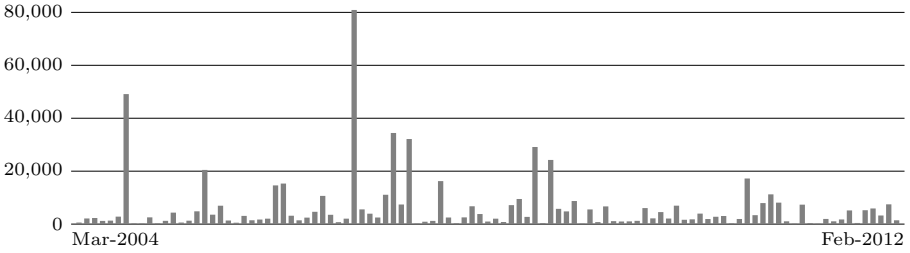


Fig. 1. Size distribution of AFP entries in lines of proof, sorted by submission date

the pervasive system-level verification of Verisoft [2], and the operating system microkernel verification in the L4.verified project [11]. CompCert 1.9.1 measures 101,608 lines with 3,723 lemmas in 143 theory files, the L4.verified repository currently contains around 390,000 lines with 22,000 lemmas in 500 theories, and the Verisoft project has published about 500,000 lines with 8,800 lemmas in over 700 theories in its three largest releases (duplicates removed). The L4.verified proofs take 8 hours to check, the Verisoft proofs an estimated 12 hours.

When scale increases to this order of magnitude, there is a phase change required in the level of support from the underlying proof assistant. Questions of knowledge management, team interaction, scalability of the user interface, performance, and maintenance become more important. In some cases, they can become more important than the presence of advanced reasoning features.

Large-scale developments concern multiple people over multiple years, with team members joining and leaving the project. The key difference between a large-scale proof and a small one, or even a multi-year PhD, is that, in the former, no single person can understand all details of the proof at any one time.

In this paper, we examine the consequences of this difference against the background of our combined experience from the L4.verified [11] and Verisoft [1,2] projects. We determine which main challenges result from the large-scale nature of these projects for the theorem proving tool, for the proof itself, and for its management. They range from proof exploration for new project members, over proof productivity during development, to proof refactoring for maintenance, and the distribution of a single proof over a whole team in managing the project. We comment on our experience addressing these challenges and suggest some solutions. We describe one of these solution in more technical detail: the proof refactoring tool *Levity*, developed in L4.verified.

Many of the issues we list also occur for projects of much smaller scale and their solution would benefit most proof developments. The difference in large-scale proofs is that they become prohibitively expensive.

While both projects used Isabelle/HOL, we posit that the challenges we identify are more general. Some of them even are already sufficiently met by Isabelle/HOL. We include them here because the primary aim of this paper is to give theorem prover developers a resource for supporting large proofs that is well founded in practitioner’s experience.

2 Challenges

We divide the description of challenges into four perspectives: a new proof engineer joining the project, an expert proof engineer interacting with colleagues during main development, the maintenance phase of a project, and the social/management aspects of a development.

2.1 New Proof Engineer Joins the Project

Over the course of a large project, team members come and go, and must be brought up to speed as quickly as possible. Learning how to use Isabelle was not an issue, because extensive documentation and local experts in the system were available. Understanding the subject of the verification, e.g. an OS kernel, was harder, but many solutions from traditional software development applied.

However, quickly inducting new proof engineers, even expert ones, into a large-scale verification remains challenging. Consider the theory graph of L4.verified in Fig. 2, which shows roughly 500 theories. As mentioned, these contain 22,000 human-stated lemmas. Adding those generated by tools takes the count to 95,000. In order to perform her first useful proof, a new proof engineer must find a way through this jungle, identify the theorems and definitions needed, and understand where they fit in the bigger picture.

At the start of L4.verified, we developed a *find_theorems* tool which allows, amongst other features, pattern-matching against theorems and their names, filtering rules against the current goal, and ranking by most accurate match. It also includes an *auto-solve* function to warn users if they restate an existing lemma; a common occurrence not only for newcomers.

Both tools were incorporated into Isabelle/HOL, but our experience echoed that of Verisoft who found that *Isabelle’s built in lemma search assumes that theories are already loaded.*

This is rarely the case in large developments like ours [...] [1]. Theorems not already loaded are not visible to the prover. We therefore extended *find_theorems* with a web interface and combined it with our nightly regression test. Any team member can search recent global project builds from their browser. Additionally, we tag Isabelle heaps [2] with revision control information and show this along with the results.

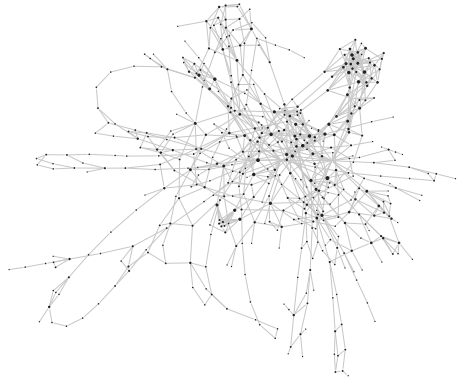


Fig. 2. Theory dependencies in L4.verified

¹ A heap is a file containing a memory dump of prover data structures. It can be loaded to avoid reprocessing theory files from scratch. Isabelle’s architecture precludes the separate compilation of theory files possible in some other provers.

In addition to theorems, provers allow custom syntax definitions, which raises questions such as *what does $[x, y. e. z]$ mean?* To a newcomer it may not even be clear if the syntax is for a type, an abbreviation, or a constant. Answering such questions requires locating where a symbol is defined. Inspired by Coq, we developed a *locate* command which identifies symbol definitions and syntax. However, the implementation requires explicit knowledge of all definition packages in Isabelle, which are user-extensible. The recent, more deeply integrated approach of the Isabelle prover IDE (PIDE) [20] is promising. In general, better prover IDEs would be beneficial, but designing them for scale is tricky. A short delay in a small test case could translate to a 30 minute wait at scale.

Ultimately, technology can solve only part of this problem. Good high-level documentation of key project areas and ideas like an explicit mentor for each new proof engineer are as important as good prover support.

2.2 Expert Proof Engineer during Main Development

For a long-term project member in the middle of proof development, the crucial issue is productivity. In this section, we highlight related challenges.

Automation. A simple way to improve productivity is automation. Domain-specific automation is important in any large-scale verification and the underlying prover should provide a safe extension mechanism. In Isabelle, the LCF approach [9] enabled both projects to provide their own specialised tactics and automated proof search without the risk of unsoundness. An example from Verisoft [2] is a sound, automated verification-condition generator translating Hoare triples about code in a deep embedded language into proof obligations in higher-order logic [17]. L4.verified developed between 10 and 20 smaller automated tactics [6, 22], including a tool to automatically state and prove simple Hoare triples over whole sets of functions.

Non-local Change. The usual mode of interactive proof is iterative trial and error: edit the proof, let the prover process it, inspect the resulting proof goals, and either continue or go back and edit further. Verification productivity hinges on this edit-check cycle being short. For local changes, this is usually the case. However, proof development is not always local: an earlier definition may need improvement, or a proof engineer would like to build on a colleague’s result. Re-checking everything between a change and the previous point of focus can take several hours if the distance between the points is large enough.

In addition, changes are often speculative; the proof engineer should be able to tell quickly whether a change helps the problem at hand, and how much of the existing proof it breaks. For the former, Isabelle provides support for interactively skipping proof content. For the latter, L4.verified implemented a proof cache that determines if a lemma remains provable, albeit not if the current proof

script will still work. The cache works by keeping track of theorem and definition statements and dependencies. A theorem remains provable if none of its dependencies changed, even though global context may have. The cache omits the proof for such theorems using Isabelle’s existing skipping mechanism, but replays all others. However, even just scanning definitions and lemma statements without proof can still take tens of minutes. Both questions could be answered concurrently: while the proof engineer turns attention to the previous focus without delay, skipping over intermediate proofs, the prover could automatically replay these proofs and mark broken ones in the background.

Placing Lemmas. As argued, proof development is non-linear. Frequently, general lemmas about higher-level concepts are required in a specialised main proof. Once proven, the question arises where to put such a more general lemma in a collection of hundreds of theory files. The answer is often not obvious, so Verisoft developed the *Gravity* tool [2] that gives advice based on theorem dependencies. But even if proof engineers have a reasonable idea where to put a lemma, long edit-check cycles will inhibit them from placing it at the correct position, leading to a significant risk of duplication and making it hard to build on results from other team members. L4.verified addressed the question of lemma movement with the *Levity* tool that we describe in [Sec. 3](#).

Avoiding Duplication. It is already hard to avoid duplication in a small project but it becomes a considerable challenge in large-scale proofs. Clearly, proving the same fact multiple times is wasteful. Despite theorem search and related features, duplication still occurred. Even harder than spotting exact duplication is avoiding similar formulations of the same concept. These will often only be recognised in later proof reviews. Once recognised, eliminating duplication can still cause large overhead. In these cases, it would be beneficial if lemmas or definitions could be marked with a deprecation tag or a comment that becomes visible at the use point, not the definition point. A further aspect is generalisation. Often lemmas are proved with constants where variables could instead have been used, which again leads to duplication. Trivial generalisation could be automated.

Proof and Specification Patterns. In software development, design patterns have been recognised as a good way to make standard solutions to common problems available to teams in a consistent way. An observation in both projects was that proof engineers tended to reinvent solutions to common problems such as monadic specifications in slightly different ways. Worse, re-invention happened even within the project. It is too simplistic to attribute this to a lack of documentation and communication alone. In software development, sets of standard solutions are expected to be available in the literature and are expected to be used. In proof and specification development, this is not yet the case. A stronger community-wide emphasis on well-documented proof and specification patterns could alleviate this problem.

Name Spaces. Names of lemmas, definitions, and types, syntax of constants, and other notational aspects are a precious resource that should be managed carefully in large projects. Many programming languages provide name space facilities, provers like Isabelle provide some mechanisms such as locales, but have mostly not been designed with large-scale developments in mind. The challenge is of the same quality but harder for provers than for programming languages: there exists a larger number of contexts, syntax and notation is usually much more flexible in provers (as it should be), and it is less clear which default policies are good. For instance, names of definitions and their syntax should be tightly controlled to define interfaces. On the other hand facts and lemmas should be easily searchable and accessible by default. At other times, the interface really is a key set of lemmas whereas definitions and notation matters less. Any such mechanism must come with high flexibility and low overhead. For instance, Mizar’s [15] requirement of explicit import lists would make large-scale program verification practically infeasible. On the other hand Isabelle’s global name space pollutes very quickly and requires the use of conventions in larger developments.

Proof Style. Isabelle supports two styles of reasoning that can be mixed freely: the imperative style, which consists of sequential proof scripts, and the more declarative style of the Isar language [19], which aims at human-readable proofs.

The greater potential for readability ostensibly is an advantage of Isar. However, achieving actual readability comes with the additional cost of restructuring and polishing. Neither of the projects were prepared to invest that effort. Instead, both styles were used in both projects, and we did not observe a clear benefit of either style. During maintenance, Isar proofs tend to be more modular and robust with respect to changes in automation. However, by that same declarative nature they also contain frequent explicit property statements which lead to more updates when definitions change.

The usually beneficial modular nature of Isar had a surprising side-effect: proof engineers inlined specialised facts within large sub-proofs. Though that is often a good idea, it also often turned out that such facts were more generally useful than initially thought, but by inlining, they were hidden from theorem search tools. Since the main mechanism for intermediate facts in the imperative style is a new global lemma statement, this occurred less frequently there.

Our recommendation is to mix pragmatically, and use the style most appropriate for the experience of the proof engineer and sub-proof at hand.

2.3 Proof Maintenance

As in software development, proof maintenance does not often get up-front attention. Fig. 3 gives an outline of proof activity over time on one L4.verified module: a main development period, multiple cleanups and bursts of activity, and a long maintenance phase, characterised by low levels of activity, starting well before the project’s end. Over time, maintenance becomes the dominant activity.

Refactoring. A large part of maintenance involves refactoring existing proofs: renaming constants, types, and lemmas; reformulating definitions or properties

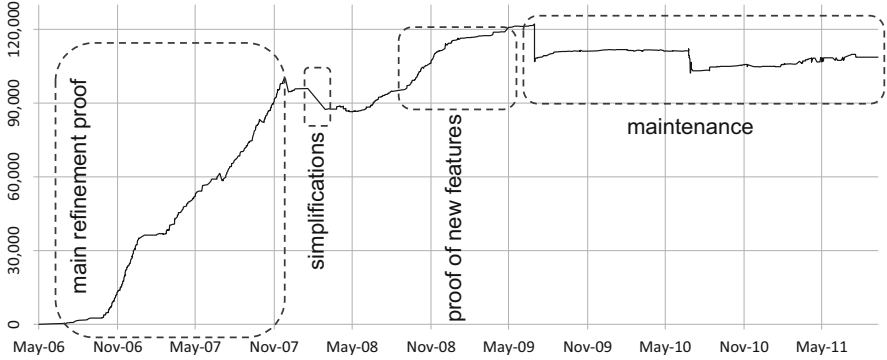


Fig. 3. Lines of proof over time in one L4-verified module

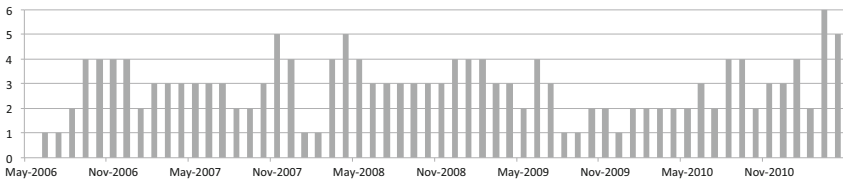


Fig. 4. Number of team members working on one L4-verified module over time

for more consistency; moving lemmas; disentangling dependencies; removing duplication. Such refactorings are in part necessitated by failures to avoid duplication during development. This is inevitable in large multi-year projects.

We have already mentioned proof refactoring tools above. Such tools are becoming popular in programming environments, but even there they are often imperfectly implemented. Current theorem proving systems offer no native support for even simple refactorings such as renaming or lemma movement. While in programming languages strict semantics preservation is paramount, it is less of an issue for proof assistants: the theorem prover will complain if the new proof breaks. In practice, time overhead is more important. Refactorings are typically non-local and can easily lead to edit-check cycles of multiple hours. This can be mitigated to a degree by reliably automating refactorings in offline nightly batch runs. Proof refactoring is an open, technically challenging research area, important for large-scale proof. Even simple renaming requires a deep semantic connection to the prover. Again, approaches like the PIDE [20] are promising, but create their own challenges in scaling to hundreds of large theories.

Debugging and Performance. A frequent maintenance issue is proofs breaking after an update elsewhere in the project. In large proofs not written for readability, finding and repairing the cause of the breakage is challenging. Conventions for maintainable use of automated tactics help, but are not sufficient.

We have found it useful to provide single-step versions of our automated tactics to help users pinpoint where proof searches go in unexpected directions. Extending this to a general design principle for all tactics would increase maintenance productivity, as well as helping beginners to understand what is happening under the hood. The same single-stepping analysis could be used to improve performance of proofs where automated methods run for too long. The right trade-off is important. For instance, Isabelle provides a tracing facility for term rewriting, but it easily overwhelms the user with information.

Context. Mathematical proof depends on context, i.e. a series of global assumptions and definitions under which one proves statements. Isabelle in particular has fine-grained context management that both projects could benefit from. However, proof context in Isabelle also determines which facts automated reasoners may use. On one hand, the concept of a background context has low mental overhead for the user if set up correctly. On the other hand, it introduces an extra-logical dependency on proof scripts. The same script might not work in a different position where the same facts are known, because the background context for automated tactics may differ. This makes it hard to move lemmas. Provers such as HOL4 [18] name such tactic contexts explicitly. This is not ideal either, since remembering their names within hundreds of theory files can be overwhelming. The Isabelle sledgehammer tool [5] provides an interesting solution: it searches in the global context, but explicitly indicates which facts were used in the proof, so that the proof now only depends on explicitly named entities. In theory this approach could be adopted for all proof search tools, making them context independent while possibly improving replay performance by reducing search. It would also answer a question frequently asked by new users: *which lemmas did automated method x use?*

2.4 Social and Management Aspects

Since proofs at the L4.verified and Verisoft scale are necessarily a team effort, social and management aspects play an important role. In this section, we concentrate on aspects that are either specific to proofs, or would not commonly be associated with proof development.

The main challenge is managing a team of proof engineers in a way which enables them to concurrently work towards the same overall proof goal with minimal overhead and duplication of work.

Discipline. In our experience, self-discipline alone is not sufficient for enforcing conventions and rules, be it for lemma naming, definitions, commit messages, or documentation. While self-discipline is a passable short-term (months) measure, adherence to conventions will deteriorate without incentive, explicit policing, or mechanical tests, especially as team members join and leave. We found that within the same group, this effect is stronger for proof development than for code. We speculate that two factors play a role: firstly, proof engineers get used to the theorem prover mechanically checking their work with immediate feedback for right and wrong; and secondly, theorem proving involves many concepts which

need names. Given many similar concepts, labelling each lemma or definition with a useful name is surprisingly hard: it can take longer to think about a lemma’s name than it takes to prove it. An effective renaming tool would allow the rectification of poor naming decisions and retroactive enforcement of conventions.

The Pragmatic Prover. An interesting challenge in managing a large-scale verification is striking the right balance between doing the work and developing tools to automate it. Duplication is a similar challenge. While duplication and proofs by copy and paste lead to obvious overhead, avoiding it can be arbitrarily hard. Both projects spent resources on increasing proof automation. In our experience, the views of the pragmatic programmer [10] are often directly applicable to proof: avoid duplication by automation wherever useful. Semi-regular reviews looking for automation opportunities can make a team significantly more productive. Semi-regular, informal code and proof reviews in general showed the same advantages as in usual software development of increased quality and cohesion within the team. In L4.verified, they were conducted mostly early in the project and in the later maintenance phase when new members joined the team. We should probably have held them more frequently.

The State of the Proof. In a large-scale project it is easy to lose view of which parts of the proof currently work, which are under development, and which are broken. A nightly and/or continuous regression test can generate and display this information easily, as long as the prover provides a scriptable batch mode.

Concurrent Proof. A key technical challenge in large-scale formal verification is efficient and useful distribution of proof sub-tasks to team members. Fig. 4 shows the number of L4.verified team members contributing per month to the module mentioned in Fig. 3. Up to six people worked on the same proof during development and maintenance. Ideally this is achieved by a compositional calculus that allows clean, small interfaces between sub-tasks, which are defined once and remain stable. In reality, this is rarely the case. Full compositionality often comes with a high price in other aspects of complexity, while interfaces are rarely small or stable. Nevertheless, work towards a common theorem could be distributed effectively in L4.verified by compositionality under side-conditions. Team members could work towards the same property on different parts of the system, or on different properties of the same part. They did not need detailed knowledge about the other properties or parts of the system [6]. When these side-conditions changed, effectively communicating them entailed an overhead. Such overhead is inevitable when for instance an invariant is discovered incrementally by the entire team. The faster the discovery, the smaller the overhead.

Distributing a proof over a team creates another management issue: avoiding that the separate pieces drift apart. An easy way of addressing this is to state the final top-level theorem first, and to use a continuous regression test to check that the parts still fit. Continuous integration avoids unpleasant surprises at the end of the project, such as discovering that an overly weak statement in one sub-proof requires significant rework in another. Note that proof interfaces do

not always have to fit together perfectly. Software patterns like adapters and bridges can be applied to proofs, and, with suitable conversion lemmas, notions in one formalism can often be transferred into another. As always, there is a trade-off: the more bridging, the higher the overhead and the harder the proof becomes to maintain.

Active Community. An active developer and user community around the main verification system is crucial. The absence of good documentation, online discussion, and fixes for problems could turn small annoyances into major show-stoppers. Active development also has a price, however. For instance, new proof assistant releases will not be fully backwards compatible—it would be detrimental if they were. Updating the proof base to the next prover release can add significant overhead. With one or two prover releases a year, a four-year project may confront a significant update 4–8 times. The L4.verified project invested in an update roughly once a year. Verisoft decided to stay with the 2005 Isabelle version, but set aside budget for back-porting important features of new releases. The latter approach works well for projects with a definite termination date. The former is more appropriate if the project goes into a longer maintenance period. A similar problem occurs in improving project- and domain-specific automation.

Libraries. Proof libraries play a role in any large-scale verification. The technical side of library development is handled well by mature theorem proving systems. Both projects made use of existing libraries and contributed back to the community. However, internal libraries are less polished and can become hard to manage. This occurs when the library is not explicitly maintained and serves only to accumulate roughly related lemmas. Sometimes this is appropriate, but producing a coherent and re-usable library requires a different approach. Good libraries do not emerge automatically, since library development takes significant effort and is usually not a key goal of the project. Verisoft members introduced the idea of a librarian [2], a person responsible for the consistency and maintenance of a particular library, allowing other team members to contribute lemmas as needed. While this alone is not sufficient to achieve a well-designed library, it does yield a much higher degree of usefulness and re-use. If possible, team leaders should set aside explicit budget for library and tool development. For both projects, the decision to do so resulted in increased productivity.

Intellectual Property. Intellectual property (IP) is not a usual aspect of formal proof, be it copyright and licenses, patents, or non-disclosure agreements. Larger projects with an industrial focus are likely to involve such agreements, which may constrain the proof. For instance, an agreement may require that proof scripts related to a particular artefact be a partner’s property, but not general libraries and tools. Classifying and separating each lemma according to IP agreements during development would be detrimental to the project. In L4.verified, automated analysis of theorem and definition dependencies identified proof parts specific to certain code artefacts, which were isolated with semi-automated refactoring.

3 Tool Support for Moving Lemmas

In this section, we describe the design of a tool we developed to solve the lemma placement issue introduced in [Sec. 2.2](#). The tool is called *Levity*, it was named for the idea that lemmas should float upward through a theory dependency graph to the position that maximises the potential for their reuse.

The fact that we developed a tool for cutting-and-pasting lemmas attests to one of the practical differences between small verification projects and larger ones. In a large development, reprocessing intervening files after moving a lemma to a new theory file may take tens of minutes or even hours; a loss of time and a distraction. For this reason, proof engineers in the L4.verified project resorted to adding comments before certain lemmas, like (`* FIXME: move *`) or (`* FIXME: move to TheoryLib_H *`), with the hope of those lemmas being moved afterward. It turns out that moving lemmas involves more than just placing them after the other lemmas that they themselves require. A broader notion of theory context is required. For instance, lemmas can be marked with `[simp]`, at or after their declaration, to add them to the global set of lemmas that are automatically applied by the simplification method.

We think that a description of the design of *Levity* and the problems we encountered will be instructive for developers of other tools that manipulate interactive proof texts.

3.1 Design and Implementation Choices

Our first idea for implementing *Levity* was to use a scripting language and lots of regular expressions, but we finally decided to use Standard ML and to exploit APIs within Isabelle. This allowed us (a) to rely on the existing parsing routines, which is especially important since Isabelle has an extensible syntax, (b) to easily access the prover’s state accumulated as theories are processed, and (c) to readily ‘replay’ lemmas so as to validate moves. The main disadvantage of this approach is that the APIs within Isabelle evolve rapidly. This not only necessitates regular maintenance, but poses the continual risk of obsolescence: a key feature used today may not be available tomorrow!

The second major design decision was to incorporate *Levity* into the nightly build process. Large proof developments are similar to large software developments. Proof engineers check-out the source tree to their terminals, state lemmas, prove, and commit changes back to the repository. Nightly regression tests check the entire proof development for errors. *Levity* is run directly after (successful) regression tests because an up-to-date heap is required for the extraction of lemma dependencies and the validation of lemma moves. We found that a second build is required after *Levity* runs and before committing any changes to the repository to ensure that the modified development builds without error. Having to run two full and lengthy builds is problematic because, as mentioned in [Sec. 2](#), several hours may elapse from check-out to commit, and, especially in an active development, there may be intervening commits which necessitate merging, and, at least in principle, further test builds.

In the original design, we intended that Levy run regularly and with minimal manual intervention, automatically shifting lemmas to the most appropriate theory file during the night in readiness for the next day's proving, but it may in fact have been better to introduce some executive supervision. The destination theories chosen by Levy are optimal in terms of potential reuse, that is Levy moves lemmas upward as far as possible in the theory dependency graph, but they are not always the most natural; for instance, it is usually better to group related lemmas regardless of their dependencies. While the possibility of stating an explicit destination, or even a list of explicit destinations, in 'fix me' comments helps, it may have been even better to provide a summary of planned moves and to allow certain of them to be rejected or modified.

In the remainder of this section, we discuss the four main technical aspects of the Levy implementation: working with the theorem prover parser, calculating where lemmas can be moved, replaying proofs, and working with theory contexts. Although the technical details are specific to the Isabelle theorem prover and the task of moving lemmas, other tools for manipulating proof developments likely also need to manipulate the text of theories, interact with a theorem prover, and handle theory contexts.

Parsing. Levy processes theories one-by-one from their source files. In processing a file, it first calls the lexer routines within Isabelle to produce a list of tokens. Working with lists of tokens is much less error prone than working directly with text and avoids many of the complications of working with the sophisticated and extensible Isabelle/Isar syntax (any syntactically correct theory file can be processed). This is only possible because rather than filter out comments and whitespace, as is standard practice, the lexer returns them as tokens.

Levy processes each file sequentially, maintaining a list of tokens to remain in that file, namely those not marked for movement and those that could not be moved, and a list of tokens yet to be processed. It shifts tokens from the latter onto the former until it finds a 'fix me' comment followed by a lemma. As far as concerns parsing, two further details are important. First, a lemma's name and attributes must be extracted. There are already functions within Isabelle to do this, but we found that they were not general enough to be called directly and we thus had to duplicate and modify them within our tool. Such duplications make a tool harder to maintain as the underlying theorem prover evolves and may engender errors that type checking cannot detect. Second, the last token in the move must be identified. This is slightly more difficult than it may seem as the keywords that terminate a proof (like `done`, `qed`, and `sorry`) may also terminate sub proofs. And, furthermore, the extent of a definition depends on the syntax defined by a particular package which may be defined externally to the main theorem prover. Since, in any case, we simultaneously replay potential moves to check for problems, we use feedback from Isabelle to find the token that ends a proof or definition.

The tokens comprising a successful move are appended to a list maintained for the destination theory. Levy always appends lemmas to the end of a theory file. Inserting them between other lemmas would effectively require having to

replay all theory files, both to detect the gaps between declarations and also to test moves in context. This would require more bookkeeping and take longer to run, but it would have the advantage of testing an entire build in a single pass. The tokens comprising failed moves are appended to the list of unmoved tokens.

Finally, theory files are remade from the lists of associated tokens, which works marvellously as Isabelle’s command lexing and printing routines are mutually inverse.

Calculating Lemma Destinations. Before trying to move a lemma, Levity calculates the lemmas, and thereby theories, on which the proof depends. They are used to validate explicit destinations, and, when none are given, to choose the destination.

For calculating lists of lemma dependencies, we modified a tool developed within the Verisoft project, called *Gravity* [2], to handle sets of lemmas, to account for earlier moves, and to handle lemmas accessed by index. This tool works through a proof term—the low-level steps that construct a new lemma from existing lemmas, axioms and rules—to construct a dependency graph.

The calculation of destination theories in *Gravity* takes dependencies on lemmas into account, but not dependencies on constant definitions. Although not usually a problem in practice, as most proofs will invoke theorems about constants in the lemma statement, detecting such problems is difficult because constants moved before their definitions are interpreted as free variables.

Controlling Isabelle. When moving a lemma, respecting its dependencies is not enough. Besides the set of theorems, proofs also depend on several other elements of the theory context; for example, syntax declarations, abbreviations, and the sets of theorems applied by proof methods like simplification. Rather than try to determine all such dependencies in advance, we decided to simply try replaying lemmas in new contexts. The tool effectively replicates the actions of a proof engineer who, after copying-and-pasting a proof, must test it interactively.

Levity replays lemmas by parsing tokens, using Isabelle library routines, into commands which are applied through *Isar*, the Isabelle theory language. *Isar* operates as a state machine with three main modes: *toplevel*, *theory*, and *proof* [19, Fig. 3.1]. For each move, Levity initially executes a command to put *Isar* into *theory* mode, the first command drawn from the list of tokens being replayed should then cause a transition into the *proof* mode; if it does not, the move fails. Subsequent commands are then passed one-by-one to *Isar*, while monitoring the state for errors and the end of a proof. We use a time-out mechanism to interrupt long running proof steps to recover from divergences in automatic tools due to differences in the original and proposed theory contexts.

Lemmas are replayed at the end of destination theories, but theories cannot be extended after they have been closed, so Levity creates a new ‘testing’ theory for each replay. Each testing theory only depends on either the proposed destination theory, or on the last successful testing theory for that destination (to account for previous moves). In general, it is necessary to import, directly or transitively, the testing theories of all required lemmas that have also been moved.

Theory Context. Interactive theorem proving inevitably involves the notion of context or state. At any point in a theory, there are the set of existing lemmas and definitions, syntax definitions, abbreviations, and sets of lemmas used by proof methods. Furthermore, Isabelle also has *locales* for fine-grained context management; lemmas stated within a locale may use its constants, assumptions, definitions, and syntax. Levity does not resolve all problems related to theory context, but it does address some aspects of locales and proof method sets.

To handle locales in Levity, the tokens being processed from a source file are fed through a filter that tracks the commands that open and close contexts² and maintains a stack of active declarations. An alternative would be to replay all commands through Isabelle and then to query the context directly. This would be more reliable but also slower. In any case, given this contextual information, Levity knows when a lemma is being moved from within a locale and it inserts tokens into the lemma declaration to re-establish the target context. The context stack also enables Levity to form fully-qualified lemma names.

As mentioned earlier, lemmas may be marked with *attributes*, like `[simp]`, to add them to proof method sets. Moving attributes with a lemma may interfere with subsequent applications of methods in other proofs. To avoid this, Levity parses attributes, strips certain of them, and inserts them as declarations at the original location. In general, such declarations, whether inserted by Levity or manually, should be tracked and considered when moving other lemmas. Consider, for example, lemmas l_1 and l_2 , both marked for movement, l_1 having the `simp` attribute, and l_2 involving a simplification that implicitly uses l_1 . Moving l_1 leaves the `simp` attribute in place. But now, moving l_2 may fail as the simplification step no longer implicitly uses l_1 .

3.2 Experience and Related Work

We ran Levity several times against the main L4.verified project development during its final months. The results were encouraging, but several problems and limitations inhibited its permanent introduction into our build process. Besides the challenges of long build times, our biggest problems were unexpected dependencies and changes to libraries within Isabelle.

By unexpected dependencies, we mean that some lemmas became ‘stuck’ at seemingly inappropriate theories, and that other dependent lemmas then became queued after them³. Levity logs lemma dependencies before moving lemmas, which helps to understand why destinations were chosen, but then it is too late to do anything. When a lemma is not moved as far as expected due to dependencies on related lemmas, it is not uncommon that those related lemma should also have been marked for movement. Ideally then, some kind of manual review should be incorporated into the process; perhaps gathering information at night while performing approved moves, then requesting new approvals during the day.

² Namely, `context`, `locale`, `class`, `instantiation`, `overloading`, and `end`.

³ When a lemma move fails, other dependent lemmas are not moved either.

Levity relies on internal Isabelle routines for parsing, analysing lemma dependencies, and interacting with Isar. When we went to prepare a public release for the latest version of Isabelle, we found that the interfaces to these routines had changed considerably from the version used in the L4.verified project, and, in fact, several essential features were no longer available. Careful compromise is needed between the evolution of a theorem prover’s design and the availability of up-to-date third-party tools (and books and tutorials); both being important factors in the success of large verification projects.

Our inability to upgrade the ML version of Levity led to a rewrite [16] using the new PIDE interface to Isabelle [20]. This approach should make the tool more robust to changes within Isabelle. But we cannot yet comment on the efficacy of this tool against a large proof development like the L4.verified project: in particular, on its integration into nightly regression tests, and the effect of asynchronous recalculations while making automatic changes to large proofs. Development continues on this new version.

Whiteside et al. [21] address the subject of proof refactoring formally and in some generality. In particular, they define a minimal proof language, its formal semantics, and a notion of statement preservation. They then propose several types of refactoring, including renaming and moving lemmas (the latter defined as a sequence of ‘swap’ operations), define some of them and their preconditions in detail, and reason about their correctness. They explicitly do not *cover all aspects of a practical implementation*, and, in contrast to Isar, their proof language does not include proof method sets and attributes, definitions, locales, or imported theory dependencies. Nevertheless, we find such a formal approach a promising way to understand and validate refactoring tools like Levity; not just to show that they preserve correctness but also that they do not introduce build failures.

4 Summary

We have described challenges that are new or amplified when formal verification reaches the scale of multi-person, multi-year projects. We draw on the experience from two of the largest such projects: Verisoft and L4.verified. Many of these challenges arise from the inability of any single human to fully understand all proof aspects. Without a mechanical proof checker, such proofs would be infeasible and meaningless. For some challenges, we have sketched solutions, and for one, we have shown in more detail how it can be addressed by tool support.

The three most important lessons learnt from our verification experience are: First, proof automation is crucial because it decreases cognitive load, allowing humans to focus on conceptually hard problems. It also decreases the length of proof scripts, reducing maintenance costs. To achieve this, prover extensibility is critical and needs to allow for custom automation while maintaining correctness. Second, introspective tools such as *find_theorems* gain importance for productivity because effective information retrieval is necessary in an otherwise overwhelming fact base. Third, proof production at large scale hinges on an

acceptably short edit-check cycle; any tool or technique that shortens this cycle increases productivity, even if temporarily sacrificing soundness.

Acknowledgements. Thomas Sewell and Simon Winwood contributed to the requirements for the Levity tool of [Sec. 3](#). We also thank June Andronick for commenting on a draft of this paper.

References

1. Alkassar, E., Hillebrand, M., Leinenbach, D., Schirmer, N., Starostin, A.: The Verisoft Approach to Systems Verification. In: Shankar, N., Woodcock, J. (eds.) VSTTE 2008. LNCS, vol. 5295, pp. 209–224. Springer, Heidelberg (2008)
2. Alkassar, E., Hillebrand, M., Leinenbach, D., Schirmer, N., Starostin, A., Tsyban, A.: Balancing the load — leveraging a semantics stack for systems verification. JAR: Special Issue Operat. Syst. Verification 42(2-4), 389–454 (2009)
3. Appel, K., Haken, W.: Every map is four colourable. Bulletin of the American Mathematical Society, 711–712 (1976)
4. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq’Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series. Springer (2004)
5. Böhme, S., Nipkow, T.: Sledgehammer: Judgement Day. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 107–121. Springer, Heidelberg (2010)
6. Cock, D., Klein, G., Sewell, T.: Secure Microkernels, State Monads and Scalable Refinement. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLS 2008. LNCS, vol. 5170, pp. 167–182. Springer, Heidelberg (2008)
7. Gonthier, G.: A computer-checked proof of the four colour theorem (2005), <http://research.microsoft.com/en-us/people/gonthier/4colproof.pdf>
8. Gonthier, G.: Formal proof — the four-color theorem. Notices of the American Mathematical Society 55(11), 1382–1393 (2008)
9. Gordon, M.J.C., Milner, R., Wadsworth, C.P.: Edinburgh LCF. LNCS, vol. 78. Springer, Heidelberg (1979)
10. Hunt, A., Thomas, D.: The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley, Reading (2000)
11. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: 22nd SOSOP, Big Sky, MT, USA, pp. 207–220. ACM (October 2009)
12. Klein, G., Nipkow, T., Paulson, L.: The archive of formal proofs (2012), <http://afp.sf.net>
13. Leroy, X.: Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In: Morrisett, J.G., Jones, S.L.P. (eds.) 33rd POPL, Charleston, SC, USA, pp. 42–54. ACM (2006)
14. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
15. Rudnicki, P.: An overview of the MIZAR project. In: Workshop on Types for Proofs and Programs, pp. 311–332. Chalmers University of Technology, Bastad (1992)
16. Ruegenberg, M.: Semi-automatic proof refactoring for Isabelle. Undergraduate thesis, Technische Universität München (2011)

17. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. PhD thesis, Technische Universität München (2006)
18. Slind, K., Norrish, M.: A Brief Overview of HOL4. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 28–32. Springer, Heidelberg (2008)
19. Wenzel, M.: Isabelle/Isar—a versatile environment for human-readable formal proof documents. PhD thesis, Technische Universität München (2002)
20. Wenzel, M.: Isabelle as Document-Oriented Proof Assistant. In: Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.) MKM 2011 and Calculemus 2011. LNCS, vol. 6824, pp. 244–259. Springer, Heidelberg (2011)
21. Whiteside, I., Aspinall, D., Dixon, L., Grov, G.: Towards Formal Proof Script Refactoring. In: Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.) MKM 2011 and Calculemus 2011. LNCS, vol. 6824, pp. 260–275. Springer, Heidelberg (2011)
22. Winwood, S., Klein, G., Sewell, T., Andronick, J., Cock, D., Norrish, M.: Mind the Gap: A Verification Framework for Low-Level. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 500–515. Springer, Heidelberg (2009)

Semantic Alliance: A Framework for Semantic Allies

Catalin David, Constantin Jucovschi,
Andrea Kohlhase, and Michael Kohlhase

Computer Science, Jacobs University Bremen
<http://kwarc.info>

Abstract. We present an architecture and software framework for semantic allies: Semantic systems that complement existing software applications with semantic services and interactions based on a background ontology. On the one hand, our **Semantic Alliance** framework follows an invasive approach: Users can profit from semantic technology without having to leave their accustomed workflows and tools. On the other hand, **Semantic Alliance** offers a largely application-independent way of extending existing (open API) applications with MKM technologies. The **Semantic Alliance** framework presented in this paper consists of three components: i.) a universal semantic interaction manager for given abstract document types, ii.) a set of thin APIs realized as invasive extensions to particular applications, and iii.) a set of renderer components for existing semantic services. We validate the **Semantic Alliance** approach by instantiating it with a spreadsheet-specific interaction manager, thin APIs for LibreOffice Calc 3.4 and MS Excel'10, and a browser-based renderer.

1 Introduction

A major research interest in the field of Mathematical Knowledge Management (MKM) consists in the development of semantic technologies: they can be prototyped on mathematical knowledge and documents where meaning is well-understood and then be transferred to other domains, where meaning is less clearly given. These semantic technologies are frequently realized in standalone applications (e.g. [Mata](#); [Math](#); [Act](#); [Cin](#)). The advantage is obvious: a standalone system can be designed autonomously without interoperability constraints towards other systems or users' previous practices with other systems. Thus, with 'no' compromises, standalone MKM technologies are typically very specialized in the kind of service they offer and very good at that.

The main disadvantage of standalone MKM systems is that they are insular. On a conceptual level, workflows of users are centered around goals to be achieved. Therefore, if the main goal is not the one solved by the standalone system, then systems must be switched, so that the insularity of standalone systems disturbs the workflow. On a technical level, the insularity often results in interoperability issues with other systems, augmenting the disturbance of a user's workflow

e.g. by the necessity of explicitly reentering data into the new structure of another system. These effects are aggravated by cognitive issues: important context information and thus understanding may get lost when switching systems: [Joh10] shows that even a focus change on a small laptop screen flushes information from short memory. All of these issues create a gap between standalone systems and other parts of the information infrastructure and workflows, and conspire to keep potential users from adopting standalone MKM systems.

In our earlier research on the adoption issue (cf. Semantic Authoring Dilemma; see [Koh05]), we have argued for the creation of *invasive semantic technology*, i.e., a technology, where the semantic services are embedded in (figuratively: invade) the host application, such that the services can draw on users' previous knowledge and experience with well-known authoring tools. This approach was inspired by HCI-driven “directness” requirements in [HHN85] and re-use ideas from Software Engineering in [Abm03]. To this end we developed the MS Excel'03 add-in “SACHS” (e.g. [KK09a]) as a semantic extension of a spreadsheet document.

In our latest project “SissI” (Software Engineering for Spreadsheet Interaction) we now want to transfer the semantic functionalities in SACHS to more spreadsheet systems: Spreadsheet users are locked into different applications like LibreOffice Calc, MS Excel, or Google Docs for reasons beyond our control. But offering SACHS functionality as *invasive* technology, every new application would induce a development effort similar to the original one for SACHS, as the functionality has to be re-created in differing application contexts and programming languages. Moreover, architectural differences like the ones between desktop applications, server-based web applications or even mobile apps, require radically different solutions for accessing the background ontology or visualizing its knowledge (e.g. in a graph viewer). Differing security issues complicate the picture even further.

The central idea to address these issues is based on a combination of the Semantic Illustration architecture in [KK09b] with a new approach towards invasive design. This gives rise to an innovative framework for semantic extensions that we present in this paper. This framework realizes an “invasive” user experience not by re-implementing semantic technologies in the host system, but by letting e.g. a separate MKM system contribute semantic services and interactions to the host user interface, managed by a ‘semantic ally’. In Section 2 we first elaborate on the combination of Semantic Illustration and invasive design, followed by a presentation and discussion of the “Semantic Alliance” architecture, which allows to build semantic allies, reusing MKM components and technologies to drive down the cost.

In section 3 we validate and report on our experiences with a first implementation “Sissi” of this Semantic Alliance framework. We review related work in Section 4 with respect to (semantic) extensions of document players and compare it with the Semantic Alliance setup. In Section 5 we summarize Semantic Alliance and draft upcoming projects.

2 The Semantic Alliance Framework

First we recap the conceptual underpinnings of our Semantic Illustration approach and sketch invasive design as an efficient replacement for invasive technology. Then we combine both approaches into a framework for semantic allies: **Semantic Alliance**. Taking the standpoint of **Semantic Alliance** being a mashup enabler in Subsection 2.2, conceptual tasks and responsibilities of each component within **Semantic Alliance** are marked and explained.

2.1 Invasive Design via Semantic Illustration

In the **Semantic Illustration** [KK09b] architecture semantic technology is made fruitful by “illustrating” existing software artifacts semantically via a mapping into a structured background ontology \mathcal{O} . We consider an underlying “illustrator” and interaction manager a **semantic ally**. Semantic services can be added to an application \mathcal{A} by linking the specific meaning-carrying objects in \mathcal{A} to concepts in \mathcal{O} , that is \mathcal{A} and \mathcal{O} are connected via a **semantic link**; see [KK09b] for a thorough discussion of the issues involved.

We combine that with a new approach to software design we call **invasive design** to obtain the same effect as invasive technology does. We observe that a service \mathcal{S} feels embedded into an application \mathcal{A} if it occupies a screen-area $\mathcal{D}_{\mathcal{S}}$ that is part of the area $\mathcal{D}_{\mathcal{A}}$ originally claimed by the application itself: if the screen area $\mathcal{D}_{\mathcal{S}}$ ‘belongs’ to \mathcal{A} in this way and the service \mathcal{S} was requested by the user from within \mathcal{A} , then the user *perceives* \mathcal{S} as an application-dependent service $\mathcal{S}_{\mathcal{A}}$, see for example Chapter 1: “*We perceive what we expect*” in [Joh10]. This perception is amplified, if a service and its request refer to the local semantic objects, we speak of “**contextualized**” services if they make use of this effect (compare with, e.g. [MDD09]).

In semantic allies the semantic link between \mathcal{A} and \mathcal{O} drawn on by semantic services \mathcal{S} given by Semantic Illustration provides such a contextualization. In particular, \mathcal{S} does not need to be implemented as application-specific invasive technology. From a technological standpoint, a *thin* client (see e.g. [NYN03]) invading \mathcal{A} is sufficient to obtain the advantages of invasive technology in the eyes of the user. This means especially, that development costs can be drastically reduced. The only condition for \mathcal{S} consists in being able to strictly outsource application-dependent parts. Note that this is again an instance of the never-ending quest for the separation of content and form, therefore semantic services should not have big difficulties in adhering to this premise.

Thus, the combination of Semantic Illustration and invasive design results in a **framework for semantic allies**, which we call “**Semantic Alliance**”. In a nutshell, it has three components:

- a platform-independent semantic interaction manager “**Sally**” (as a semantic ally), that has access to contextualizable semantic services, and that
- partners with a set of invasive, thin, application-specific “**Alex_A**”, that essentially only manage user interface events in \mathcal{A} , and that

- has access to a set of application-independent screen area managers “ Theo_n ” that can render the available services.

Note that we restrict our attention here to applications which come in the form of a “document player”, i.e., whose purpose is to give the user (read/write) access to structured data collections that can be interpreted as “documents” or “collections of documents” in some way. Prime examples of this category of applications include office suites, CAD/CAM systems, and Web2.0 systems.

2.2 The Semantic Alliance Framework as a Mashup Enabler

We have observed above that functionalities of a semantic extension system can be realized with invasive design. The **Semantic Alliance** framework introduced in this paper is based on the additional observation that the pertinent semantic extensions are already largely implemented in web-based Mathematical Knowledge Management systems (wMKM), and that semantic allies can be realized by mashing up wMKM systems with the original application. But in contrast to traditional mashups, which integrate web data feeds in a broad sense into web portals, the **Semantic Alliance** framework mashes up the GUIs of wMKM systems and applications themselves. In this sense, the **Semantic Alliance** framework can be considered a **mashup enabler**, a system that transforms otherwise incompatible IT resources into a form where they can be combined.

Let us assume \mathcal{S} to be a wMKM system drawing on an ontology \mathcal{O} .¹ In the **Semantic Alliance** framework, the task of the mashup enabler between \mathcal{A} and \mathcal{S} then is split into three parts (see Figure 1).

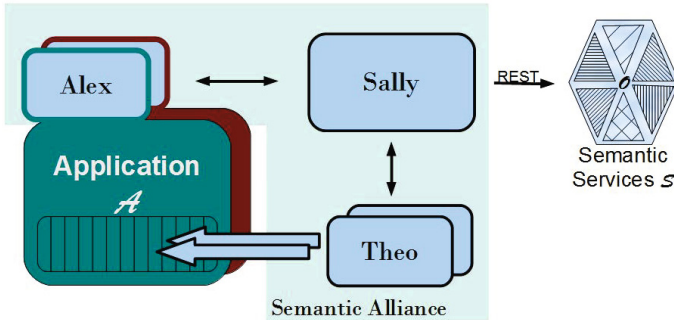


Fig. 1. Semantic Alliance as a Mashup Enabler for Semantic Allies

¹ For the sake of the exposition we assume that \mathcal{S} runs as a web service and is accessed via the Internet, but nothing hinges on this; for offline situations, \mathcal{S} could be started locally. However, since \mathcal{O} is probably the biggest investment necessary for enabling semantic services (discussed in [KK09a]) via Sally, a shared, web-accessible mode of operations for \mathcal{S} is probably the most realistic until we have distributed MKM systems.

Sally. The main part of the mashup enabler is realized in the “Sally” component which integrates the functionalities of \mathcal{A} and \mathcal{S} into a joint user interface and interaction model and thus realizes the semantic extension functionality for \mathcal{A} . Sally can draw on an implementation of abstract document types, which on the one hand abstract from the particulars of the data structures in \mathcal{A} and on the other hand tie particular interactions to (abstract) document fragments. For instance, a click on a cell in a spreadsheet should lead to a different reaction than a click to a textbox in a presentation.

Alex. The application \mathcal{A} is extended by a slim API “Alex”² that reports and executes relevant user interactions with \mathcal{A} like cell clicks in spreadsheets to and from Sally. It allows to store a **semantic illustration mapping**, i.e., a mapping between semantic objects in \mathcal{A} and concepts in \mathcal{O} , with the document itself. Note that – on a general level – any open API of an application \mathcal{A} provides us with an opportunity for invasion. Moreover, note that a thin Alex limits the dependency on the host system; in the SACHS project, where semantic functionality was directly implemented as a \mathcal{A} extension, we were confronted with an automatic Windows security update that made the used sockets inoperative, forcing major re-implementation.

Theo. To enable invasive design, a screen-area manager “Theo”³ is needed. \mathcal{S} -supplied content is embedded as “popup” into the GUI of \mathcal{A} . This embedding can be implemented at two levels: *natively*, if \mathcal{A} ’s GUI allows to embed browser layout engines or *at the operating system level* by superimposing browser windows over the GUI of \mathcal{A} . Given enough care the latter solution can be made very similar to a native integration from a user point of view.

Nowadays software applications come in three flavors: traditional desktop applications, web/browser-based applications, and mobile apps. **Semantic Alliance** can cope with all of these, if we are flexible in the deployment of the **Semantic Alliance** components:

1. For a *desktop application*, the components of **Semantic Alliance** run as local processes and use a runtime environment for a browser layout and interaction engine to provide operating-system level GUI embedding (see Figure 1).
2. For *mobile apps*, the application comes in two parts, the document and the core data structures reside on an application server \mathcal{A}_{core} , whereas (some of) the user interface functions (\mathcal{A}_{UI}) run on the mobile device in the respective app execution environment. In this situation (see Figure 2), we usually cannot assume that a local process can be started, so the **Semantic Alliance** components must run as web services on a **Semantic Alliance** server. Here, **Alex** is realized as a web service that (a) distributes the user

² Note that the **Alex** is the only real “invasive” part of **Semantic Alliance**, we have named this system after Alexander the Great; one of the mightiest invaders in history.

³ German readers may recognize, that “Theo” is a shorthand for “Karl-Theodor” as someone who pretends towards others that he does something, but in fact he’s doing something else.

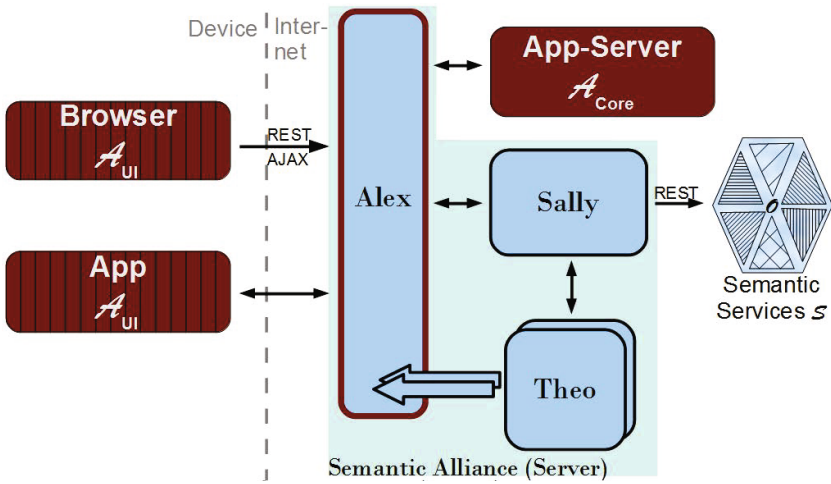


Fig. 2. Semantic Alliance for Mobile Apps

interaction events to \mathcal{A}_{core} and Sally, and (b) aggregates the contributions of \mathcal{A}_{core} (HTML pages) with those from the semantic services (via Sally and Theo) into a consistent aggregated user experience.

- For *browser-based applications*, one of the scenarios above applies: If we cannot start processes (e.g. in locked down web-kiosk-style situations) we use the “mobile apps” framework, otherwise a desktop-like deployment may have advantages. In this case, Alex is realized as JavaScript browser extension that embeds wMKM content directly into the web document used as GUI by the application server.

Note that in all these cases, the Semantic Alliance components perform the same functions and communicate with each other in essentially the same way, even if two incarnations of Semantic Alliance differ in the way they deliver their GUI contributions.

Observe that we need a two-way communication between Alex and Sally to realize even simple semantic interactions, whereas Sally and the wMKM system \mathcal{S} can restrict their communication to a client-server communication, where Sally takes the role of a client. For the browser-based case, communication between the browser and Alex as well as between Alex and the app server is unidirectional REST (Representational State Transfer). Observe also that for the “mobile apps” case, the setup in Figure 2 is only possible, if the communication between the respective app server and app execution environment relies on open APIs or open standards so that Alex can be positioned as a ‘man in the middle’. This requirement is the analogue of the “open API” requirement postulated for the desktop application case.

2.3 Building on Open APIs

The **Alex** component in our architecture is an extension of an application \mathcal{A} , which is a document player. **Alex** uses the open API (that allows business developments for \mathcal{A} from outside) to provide communication about internals of the document player.

There are multiple levels at which the open API of \mathcal{A} can offer access, in particular, these data can be separated into a content API “**contentAPI**” (the internal document object model or DOM) and a form API “**layoutAPI**” (its representation). In order to achieve Semantic Illustration, the document player needs to provide access to the DOM, while the access to representational data is needed for invasive design. For example, in a spreadsheet system, an extension would have access via the **contentAPI** to a cell, its formula and value, and via the **layoutAPI** to its width, its color and its position within \mathcal{A} ’s window frame.

Therefore, we consider an open API of a document player suitable to be used in **Semantic Alliance**, if the open API includes a **contentAPI** as well as a **layoutAPI**. Only with such an API can the semantic ally provide contextual information. Note that “contextual” comprises two dimensions: It is contextual from a semantic point of view (the semantic illustration mapping ties objects to ontology terms), and from an HCI perspective (using the positioning features of the **layoutAPI**, the information is displayed in a locus near the object of interest).

Interestingly, in most desktop systems (in particular spreadsheet systems), there is no clear distinction between the **contentAPI** and the **layoutAPI**. In contrast, in web systems, there is a clear separation between the document itself and its rendering. The underlying reason consists in the assumption that the document lives in the “cloud” but the rendered document in a browser. Moreover, the document player is not the browser itself, but just a browser window. In particular, due to security issues the communication language (e.g. Javascript) has no access to browser facilities. Thus, unfortunately, the **layoutAPI** for web document players is very limited or even inexistent for most of these. For **Google Docs** for example, we had a very close look — yielding that it does not provide basic **layoutAPI** features.

3 A Validation of the Semantic Alliance Framework

In order to validate the **Semantic Alliance** architecture, we have implemented it for spreadsheet systems within the **SiSSI** project. Concretely, we have realized **Alexes** for **LibreOffice Calc 3.4** [\[Lib\]](#) and **MS Excel 2010** to show the feasibility of the framework, but also to get first measures on the efficiency of the **Semantic Alliance** setup. In order to evaluate the new aspects of the **Semantic Alliance** framework, our goal was to reach feature parity with and compare the actual expenditures of doing so for two essential semantic services offered by the **SACHS** system for **MS Excel 2003** spreadsheets:

Definition Lookup & Semantic Navigation in SACHS. Here, the event of a cell click triggers the semantic interaction that the user had previously chosen. If she opted for a “**Definition Lookup**”, then SACHS displays the definition from the background ontology (associated with the selected cell via the semantic illustration mapping) as a native MS Excel popup close to the chosen cell. If the found definition involves other concepts that are semantically linked to cells in the spreadsheet, then the “**Semantic Navigation**” service (enabled in SACHS’ dependency graph display option) allows the user to navigate to the resp. cell on click of the according graph’s node.

3.1 Sissi: An Implementation of Semantic Alliance

In the following we describe the realized components of Semantic Alliance in the SiSSI project, which show the feasibility of Semantic Alliance.

Alex_A: Managing Interface Events

Our Alex instance for LibreOffice “Alex_{Calc}” uses LibreOffice’s Java open API for communication tasks as well as for exporting and managing requested UI events. In contrast, the MS Excel 2010 Alex instance “Alex_{Excel}” is solely based on .NET infrastructure (here C#).

Due to Semantic Alliance’s invasive design requirement the employment of semantic services is done from within the respective application, therefore Alex_{Calc} and Alex_{Excel} are responsible for starting and stopping Sally. Note that from the user’s perspective a semantic ally is started, so the menu entry is called “Sally”. To support multiple document types (like spreadsheets, presentations, or texts), Alex identifies itself and its document type to Sally during initialization. It also transmits to Sally the semantic illustration mapping stored with the spreadsheet document. Now, Sally as the semantic ally takes on responsibility for all semantic interactions. For the moment, the Alexes only

- report cell click events in the application \mathcal{A} to Sally together with the cell’s position information (X- and Y-coordinates in pixels) and the user’s display option (definition lookup or graph exhibition), and
- move the cursor to a cell when requested from Sally.

In particular, the Alexes indeed stay very thin. Depending on future semantic interactions based on other semantic objects, the UI listeners and UI actions have to be extended. But note that the former extensions are mere expansions of the listeners, that already are in place, and that the latter will handle nothing more complicated than the resp. semantic objects.

Sally: Merging Interfaces and Interactions

Sally is the central interaction manager of the Semantic Alliance framework and can be used universally for all kinds of semantic services. As we have seen in Figures 1 and 2, Sally comes in two flavors, one for a desktop setting and one

for a web-based application. These two only differ in the communication to their respective **Theos** and possibly **Alexes**. Concretely, we have only implemented the desktop variant for now. As **Sally** has to be cross-platform, it is implemented in Java making use of a Socket and WebSocket server for communication. **Sally** keeps an abstract model of the documents played in the application, an abstract interpretation mapping based on this, and maintains an abstract model of the UI state. These abstractions are useful so that **Sally** can communicate with different **Alexes/Theos**. To avoid hard-coding the set of services, **Sally** will query the wMKM system for the services available for the respective document types registered by the **Alexes** connected to **Sally**.

Theo: *Managing Screen-Area*

The main purpose of the **Theo** component is to provide interface items upon request by **Sally**. This component is realized as an instance of XULRunner [Xulb] — the naked layout and communication engine behind Mozilla Firefox and Thunderbird. The layout of the interface items is given in the XUL format [Xula] and the interactions are handled via JavaScript. For math-heavy applications, it is important that **Theo** allows HTML5 presentation of any interface item (buttons, text boxes, etc.) and text content. Note that **Theo** only concentrates on the rendering of interface elements, acting as a renderer for **Sally**, which sends content, placement and size information via Web Sockets⁴. Note that the communication channel is bi-directional: **Theo** events (e.g. clicking on a node in a dependency graph) are communicated to **Sally**, which then coordinates the appropriate reaction.

wMKM = Planetary: Provisioning Semantic Services

For **Sissi**, we (re-)use much of the Planetary system [Koh+11; Planetary] as the underlying wMKM system. The Planetary system is a Web 3.0 (or a Social Semantic Web) system for semantically annotated technical document collections based on MKM technologies. The background ontology \mathcal{O} for semantic illustration is stored as a collection of documents in a versioned XML database, that indexes them by semantic functional criteria and can then perform server-side semantic services. Results of these queries (usually fragments selected/aggregated from \mathcal{O}) are transformed to HTML5 via a user-adaptive and context-based presentation process.

3.2 Discussion

To gain an intuition on the runtime-behavior of the **Semantic Alliance** framework let us look at the new realization of **SACHS** functionality described earlier. In the concrete examples, we reuse the **SACHS** ontology already described in [KK09a] which could be adapted to the new setting.

⁴ The natural communication via sockets is prohibited by XULrunner for security reasons; Web Sockets provide a safer abstraction that we can use in this context.

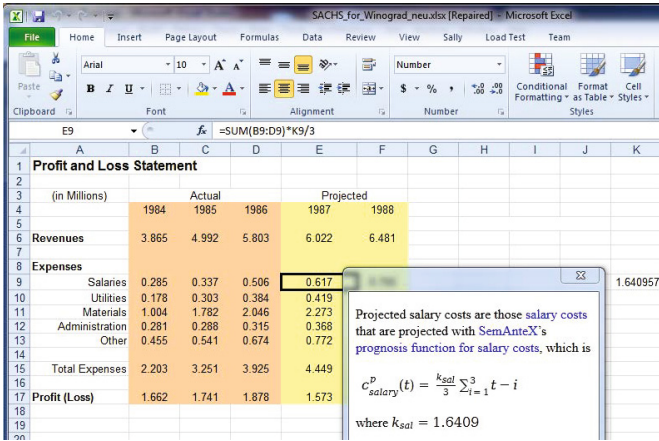


Fig. 3. Definition Lookup in Sissi

Definition Lookup & Semantic Navigation in Sissi. In the Sissi implementation (see Figure 3), Alex responds to a click of cell [E9] by requesting a **definition lookup** window from Sally, which requests an HTML5 document from the wMKM system \mathcal{S} , on whose arrival Theo overlays the \mathcal{A} -GUI at the appropriate location with a browser window containing the requested information.

The definition lookup or the dependency graph service is invoked, as of now, by the user’s selection in the “Sally menu” (visible in Fig.3 resp. 4). As this is an interaction management task, it naturally belongs to Sally’s tasks, so in the near future interaction configuration will be enabled via Sally itself.

Now, let us look at Figure 4, which presents the **dependency graph** of cell [D15], that is, the graph of “Actual Expenses per time at SemAnteX Corp”. The nodes in this graph are hyperlinks to their (via the semantic illustration mapping) associated cells. For example, the concept “Actual Utility Costs” from \mathcal{O} is associated with

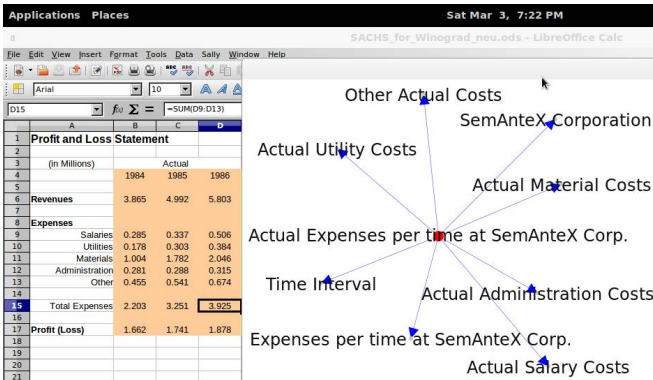


Fig. 4. Semantic Navigation in Sissi

cell range [B10:D10] on the same worksheet. If the user clicked this node, then Theo – which is in control of the window that displays the dependency graph – reports this click event to Sally, that interprets it as a navigation request

and relays this to **Alex**, which in turn moved the cursor to cell [D10] in the GUI of \mathcal{A} (based on the evident heuristic). Note that semantic navigation works workbook wide and can be extended to go across. Figure 4 is a screenshot taken from a **LibreOffice Calc** document on a Linux machine, which verifies **Sissi**'s platform- and application independence.

This means that we reached feature parity wrt. **SACHS**' definition lookup and semantic navigation as described. But even with these very limited **Alex** prototypes we already obtain some semantic features that go beyond the old **SACHS** implementation:

1. The **JOBAD** interaction framework employed in **Planetary** allows to make the interactions semantically interactive by embedding semantical services in them as well [GLR09]. For instance, if a fragment of the definition lookup text (e.g. “salary costs” in Figure 3) is linked to an ontology concept that is itself associated with a cell in this workbook (in our example, “salary costs” are associated with the cell range [B9:D9]), then semantic navigation (which was only available via concept graphs in **SACHS**) comes for free.
2. The **SACHS** system was severely limited by the fact that native **MS Excel 2003** popups are limited to simple text strings, which made layout, formula display, and interactivity impossible. In the **Semantic Alliance** framework, we have the full power of dynamic HTML5⁵ at our hands. In particular, **OpenMath** formulae can be nicely rendered (see Figure 3), which gives spreadsheet users visual (and navigational) support in understanding the computation (and the provenance) of the respective values.
3. **Planetary** has an integrated editing facility that can be used for building the background ontology as an invasive service and with a little bit more work on **Sally**, it can be used for managing the semantic illustration mapping.

Note that in the **Semantic Alliance** architecture, **Sally** is a reusable component, which implies that **Planetary** services offered via **Sally** are open to other applications \mathcal{A} as long as they have access to a suitable (thin) **Alex_A**.

To conclude our discussion let us see how our initial claim that the **Semantic Alliance** framework allows the rapid deployment of semantic services in applications holds up to the implementation experiences. Naturally, the development time of **Alex_{Calc}** cannot be considered separately as it is indistinguishable from the expenditures for developing **Semantic Alliance** and **Sissi** up to this point. But we can look at **Alex_{Excel}** and its development costs: Even though the author was not familiar with the **MS Excel .NET** backend, he succeeded building **Alex_{Excel}** in about 40 hours. Actually, much of this time was not spent on realizing the basic functionality, but on getting around idiosyncrasies of the **MS Excel** system. For instance, **MS Excel** removes the cell focus markers, when the spreadsheet window loses focus (which it necessarily does in the **Semantic Alliance** framework, since the **Theo** supplied window gets the focus, as **Theo** is a separate

⁵ The full power of dynamic HTML5 means in particular HTML+CSS for text layout, MathML for formula display, SVG/Canvas for diagrams, and JavaScript for interactivity.

process). Note that in this paper the general framework for semantic allies is presented, thus no users are yet involved, which in turn means that usability is not yet an issue. The validation of the framework consists in demonstrating its feasibility and confirming its claimed value, the nice cost-benefit ratio, which is exactly based on low API development costs.

4 Related Work

We have presented the **Semantic Alliance** framework as a mashup enabler for semantic services with already existing applications, that allows to use those services from within these applications to overcome users' potential motivational bootstrap hurdles, i.e., that yields invasive design. Here, we want to portray **Semantic Alliance**'s contribution by assessing related work. In particular, we review existing relevant (semantic) extensions of document players resp. documents and the frameworks used with respect to our architecture.

In recent years, a big variety of *mashup enablers* was created. "Greasemonkey" [Pil05] is a well-known example for a client-side extension of a web browser. In particular, it is a Firefox extension that allows to write scripts to persistently alter web pages for a user on the fly. But Greasemonkey as well as those other mashup enablers are limited to offer resulting web services. With the **Semantic Alliance** architecture a very different kind of service is enabled: an application-based service. Note that this application might be a web app, but can also be a desktop-centered component of an office suite.

We can also consider **Semantic Alliance** as a *mashup builder* like "Marmite" [WH07]. This specific one aligns programs and their data flows and is realized as a proxy server, that mediates between a web browser and a webpage. Again, there seems to be an underlying assumption that mashups only live on the web, the very thing that **Semantic Alliance** extends.

When focusing on enabling semantic services as extensions of existing programs, we observe that the number of *semantic service aggregators* is on the rise. The web search engine "WATSON" [dMT11], a 'gateway' for the Semantic Web, for a web example, collects, analyzes and gives access to ontologies and semantic data on the Web. But the potential semantic services can only be used via a unified UI, so that such service aggregators have to be considered standalone systems, which we argued against in the introduction. Therefore, our **Semantic Alliance** framework only contains the aggregating component **Sally**, but separates its tasks via **Alex** and **Theo**.

There are desktop variants of semantic service aggregators like the "Semantic Desktop" [Sem], which is an open-source semantic personal information management (PIM) and collaboration software system, that offers semantic services based on the relations of structured data of desktop applications. In particular, it uses desktop crawlers based on distinct document type- and application ontologies to collect metadata. Within the Semantic Desktop framework third party components can be integrated via pluggable adaptors. The services can be used by a user via a PIM-UI provided by the Semantic Desktop or via application-UIs that are extended by invasive technology. The cost and redundancy issues

discussed in the introduction apply here as well, that is, the development costs for each semantic extension is very high. In the **Semantic Alliance** architecture only thin, invasive add-ons (“**Alex**”es) are necessary to provide the same user experience. Moreover, the **Semantic Desktop** focuses on services using automatically gathered semantic data based on ontologies of desktop applications. In contrast, **Semantic Alliance** centers around specific documents with their individual ontology given by a semantic illustration mapping.

“truenumbers” [Tri] is a technology for supporting the representation, management and copy/pasting of engineering values as semantically enhanced data. It encapsulates for instance the magnitude and precision of a number, its units, its subject, and its context. The technology is realized as a set of plug-ins for e.g. Eclipse, MS Office, or Adobe PDF. The metadata are stored in hosted or private clouds and semantic services are offered via a Web client. We consider truenumbers to be related closely to **Semantic Alliance**. But as truenumbers only targets engineering values, its semantic objects are restricted, and hence, its scope is limited.

In the **Semantic Alliance** framework we are elaborating the idea of “Interface Attachments” [Ols+99]. These are small interactive programs that augment the functionality of other applications with a “*minimal set of ‘hooks’ into those applications*” [ibid., p. 191], where the hooks exploit and manipulate the *surface representation* of an application. We target with a **Theo** component this manipulation of the surface representation, but with our thin **Alex** component we only address the mentioned minimality of hooks, as our main goal does not consist in efficient service exploitation, but in the exploitation of the underlying semantics of a document played by an application. Note that our **Theo** is application-independent in contrast to “Interface Attachments” manipulation hook. Moreover, with one **Sally** component we make complex services available to distinct applications.

“Contextual Facets” [MDD09] are a UI technique for finding and navigating to related websites. They are built automatically based on an analysis of webpages’ semi-structured data aligned to a user’s short term navigation history and filter selection. They are contextual in the sense, that a user’s recent and current webpage elements usage determines the context for available services (here, navigation links). In **Sally**, **Semantic Alliance**’s interaction manager, we use users’ interaction with semantic objects (that are given by an abstract document type) like clicking a cell as context for semantic services. The time component of user actions is not yet integrated within **Sally**. Note that we can consider a semantic ally as a contextual facet, so that the **Semantic Alliance** architecture is a transition from the “Contextual Facets” technique from a browser document extension to general document extensions via the underlying abstract document type. We also like to mention that MEDYNSKIY ET AL. report for their implementation “FacetPatch” that “*participants in an exploratory user evaluation of FacetPatch were enthusiastic about contextual facets and often preferred them to an existing, familiar faceted navigation interface.*” [MDD09; p. 2013], which makes us hopeful for the adaptation of **Semantic Alliance**.

5 Conclusion and Future Work

We have presented the **Semantic Alliance** architecture and software framework that allows the user from within different standalone document players to use a semantic ally service (or semantic ally for short). From her point of view, she uses specialized semantic allies tailored to the respective application, whereas from a technical perspective, the main component of the framework is a single, universal semantic ally **Sally** mashing up distinct semantic services with the resp. application's GUI. This is possible due to an innovative task distribution in **Semantic Alliance** based on a combination of the Semantic Illustration architecture and invasive design.

In particular, the application-specific parts of a service are outsourced to **Alexes**, which are just responsible for managing the application's UI events and thus can be built thin. In our reference implementation for **Alex_{Excel}**, the development merely took a week. The rendering parts of a service are executed by **Theos**. **Sally**, the technical semantic ally, acts as an interaction manager between the components and services on the one side and the user on the other. As such it requires most development effort and time and incorporates thus substantial MKM technology.

Our particular **Sally** implementation in **Sissi** as of now is desktop-based and realized in Java, integrating much of the semantic functionality via web-services, and our **Theo** is browser-based. Therefore, our setup of **Semantic Alliance** is fully operating-system independent. So far we have only used **Sally** to mediate between different types of spreadsheet programs, in the future we want to exploit this to incorporate different applications with semantic allies. This will be simple for the other elements of the **LibreOffice** suite, hence, we are looking forward to blend semantic services between different document types in the near future. Moreover, work is currently under way on an **Alex** for a CAD/CAM system as envisioned in [Koh+09]. We expect the main work to be in the establishment of an abstract document model (which resides in **Sally** and is shared across semantic allies for CAD/CAM systems) and the respective background ontology.

Even though our work only indirectly contributes to the management of mathematical knowledge, we feel that it is a very attractive avenue for outreach of MKM technologies. For instance, in the **SiSsI** project for which we have developed the **Semantic Alliance** framework, we will use **Sally** to integrate verification of spreadsheet formulae against (formal) specifications in the background ontology or test them against other computational engines. In CAD/CAM systems the illustration mapping can be used to connect CAD objects to a bill of materials in the background ontology, which in turn can be used to verify physical properties (e.g. holding forces). Computational notebooks in open-API computer algebra systems like **Mathematica** or **Maple** can be illustrated with the papers that develop the mathematical theory. Theorem provers can be embedded into **MS Word**, ... — the opportunities seem endless.

The **Semantic Alliance** framework is licensed under the GPL and is available at <https://svn.kwarc.info/repos/sissi/trunk/>.

Acknowledgements. The research in the SiSiI project is supported by DFG grant KO 2428/10-1.

References

- [Act] ACTIVE MATH, <http://www.activemath.org> (visited on June 05, 2010)
- [Aßm03] Aßmann, U.: Invasive software composition, pp. I–XII, 1–334. Springer (2003) ISBN: 978-3-540-44385-8
- [Car+09] Carette, J., Dixon, L., Coen, C.S., Watt, S.M. (eds.): MKM 2009, Held as Part of CICM 2009. LNCS (LNAI), vol. 5625. Springer, Heidelberg (2009)
- [Cin] Cinderella: Interactive Geometry Software, <http://www.cinderella.de> (visited on February 24, 2012)
- [dM11] d’Aquin, M., Motta, E.: Watson, more than a Semantic Web search engine. *Semantic Web* 2(1), 55–63 (2011)
- [GLR09] Giceva, J., Lange, C., Rabe, F.: Integrating Web Services into Active Mathematical Documents. In: Carette, J., Dixon, L., Coen, C.S., Watt, S.M. (eds.) MKM 2009, Held as Part of CICM 2009. LNCS (LNAI), vol. 5625, pp. 279–293. Springer, Heidelberg (2009), <https://svn.omdoc.org/repos/jomdoc/doc/pubs/mkm09/jobad/jobad-server.pdf>
- [HHN85] Hutchins, E.L., Hollan, J.D., Norman, D.A.: Direct manipulation interfaces. *Hum.-Comput. Interact.* 1(4), 311–338 (1985) ISSN: 0737-0024
- [Joh10] Johnson, J.: *Designing with the Mind in Mind: Simple Guide to Understanding User Interface Design Rules*. Morgan Kaufmann Publishers (2010)
- [KK09a] Kohlhasse, A., Kohlhasse, M.: Compensating the Computational Bias of Spreadsheets with MKM Techniques. In: Carette, J., Dixon, L., Coen, C.S., Watt, S.M. (eds.) MKM 2009, Held as Part of CICM 2009. LNCS (LNAI), vol. 5625, pp. 357–372. Springer, Heidelberg (2009), <http://kwarc.info/kohlhasse/papers/mkm09-sachs.pdf>
- [KK09b] Kohlhasse, A., Kohlhasse, M.: Semantic Transparency in User Assistance Systems. In: Mehlenbacher, B., et al. (eds.) *Proceedings of the 27th Annual ACM International Conference on Design of Communication (SIGDOC)*, Bloomington, Indiana, USA. ACM Special Interest Group for Design of Communication, pp. 89–96. ACM Press, New York (2009), <http://kwarc.info/kohlhasse/papers/sigdoc09-emtrans.pdf>, doi:10.1145/1621995.1622013
- [Koh+09] Kohlhasse, M., Lemburg, J., Schröder, L., Schulz, E.: Formal Management of CAD/CAM Processes. In: Cavalcanti, A., Dams, D. (eds.) *FM 2009*. LNCS, vol. 5850, pp. 223–238. Springer, Heidelberg (2009), <http://kwarc.info/kohlhasse/papers/fm09.pdf>
- [Koh+11] Kohlhasse, M., et al.: The Planetary System: Web 3.0 & Active Documents for STEM. *Procedia Computer Science* 4, 598–607 (2011); Sato, M., et al. (eds.): *Special issue: Proceedings of the International Conference on Computational Science (ICCS)*. Finalist at the Executable Papers Challenge (2011), <https://svn.mathweb.org/repos/planetary/doc/epc11/paper.pdf>, doi:10.1016/j.procs.2011.04.063

- [Koh05] Kohlhasse, A.: Overcoming Proprietary Hurdles: CPoint as Invasive Editor. In: de Vries, F., et al. (eds.) Proceedings at Open Source for Education in Europe: Research and Practise, pp. 51–56. Open Universiteit Nederland, Heerlen (2005), <http://hdl.handle.net/1820/483>
- [Lib] Home of the LibreOffice Productivity Suite, <http://www.libreoffice.org> (visited on November 13, 2011)
- [Mata] Mathcad: Optimize your design and engineering, <http://www.ptc.com/products/mathcad> (visited on February 24, 2012)
- [Matb] Mathematica, <http://www.wolfram.com/products/mathematica/> (visited on June 05, 2010)
- [MDD09] Medynskiy, Y., Dontcheva, M., Drucker, S.M.: Exploring websites through contextual facets. In: Proceedings of the 27th International Conference on Human Factors in Computing Systems, CHI 2009, pp. 2013–2022. ACM, Boston (2009) ISBN: 978-1-60558-246-7
- [NYN03] Nieh, J., Jae Yang, S., Novik, N.: Measuring thin-client performance using slow-motion benchmarking. ACM Trans. Comput. Syst. 21, 87–115 (2003) ISSN: 0734-2071
- [Ols+99] Olsen Jr., D.R., et al.: Implementing interface attachments based on surface representations. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: the CHI is the Limit, CHI 1999, pp. 191–198. ACM, Pittsburgh (1999) ISBN: 0-201-48559-1
- [Pil05] Pilgrim, M.: Greasemonkey Hacks: Tips & Tools for Remixing the Web with Firefox (Hacks). O’Reilly Media, Inc. (2005) ISBN: 0596101651
- [Planetary] Planetary Developer Forum, <http://trac.mathweb.org/planetary/> (visited on September 08, 2011)
- [Sem] Semantic Desktop, <http://www.semanticdesktop.org/> (visited on February 24, 2012)
- [Tru] Truenumbers, <http://www.truenum.com> (visited on February 24, 2012)
- [WH07] Wong, J., Hong, J.I.: Making mashups with marmite: towards end-user programming for the web. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI 2007, pp. 1435–1444. ACM, San Jose (2007) ISBN: 978-1-59593-593-9
- [Xula] XUL language, <https://developer.mozilla.org/en/XUL> (visited on January 30, 2012)
- [Xulb] XULRunner Runtime Environment, <https://developer.mozilla.org/en/XULRunner> (visited on February 29, 2012)

Extending MKM Formats at the Statement Level

Fulya Horozal, Michael Kohlhase, and Florian Rabe

Computer Science, Jacobs University Bremen, Germany

<http://kwarc.info>

Abstract. Successful representation and markup languages find a good balance between giving the user freedom of expression, enforcing the fundamental semantic invariants of the modeling framework, and allowing machine support for the underlying semantic structures. MKM formats maintain strong invariants while trying to be foundationally unconstrained, which makes the induced design problem particularly challenging.

In this situation, it is standard practice to define a minimal core language together with a scripting/macro facility for syntactic extensions that map into the core language. In practice, such extension facilities are either fully unconstrained (making invariants and machine support difficult) or limited to the object level (keeping the statement and theory levels fixed).

In this paper we develop a general methodology for extending MKM representation formats at the statement level. We show the utility (and indeed necessity) of statement-level extension by redesigning the OMDoc format into a minimal, regular core language (strict OMDoc) and an extension (pragmatic OMDoc) that maps into strict OMDoc.

1 Introduction

The development of representation languages for mathematical knowledge is one of the central concerns of the MKM community. After all, practical mathematical knowledge management consists in the manipulation of expressions in such languages. To be successful, MKM representation formats must balance multiple concerns. A format should be expressive and flexible (for depth and ease of modeling), foundationally unconstrained (for coverage), regular and minimal (for ease of implementation), and modular and web-transparent (for scalability). Finally, the format should be elegant, feel natural to mathematicians, and be easy to read and write. Needless to say that this set of requirements is over-constrained so that the design problem for MKM representation formats lies in relaxing some of the constraints to achieve a global optimum.

In languages for formalized mathematics, it is standard practice to define a minimal core language that is extended by macros, functions, or notations. For example, Isabelle [\[Pau94\]](#) provides a rich language of notations, abbreviations, syntax and printing translations, and a number of definitional forms. In narrative

formats for mathematics, for instance, the $\text{T}_{\text{E}}\text{X}/\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ format – arguably the most commonly used format for representing mathematical knowledge – goes a similar way, only that the core language is given by the $\text{T}_{\text{E}}\text{X}$ layout primitives and the translation is realized by macro expansion and is fully under user control. This extensibility led to the profusion of user-defined $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ document classes and packages that has made $\text{T}_{\text{E}}\text{X}/\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ so successful.

However, the fully unconstrained nature of the extensibility makes ensuring invariants and machine support very difficult, and thus this approach is not immediately applicable to content markup formats. There, MathML3 [ABC⁺10] is a good example of the state of the art. It specifies a core language called “strict content MathML” that is equivalent to OpenMath [BCC⁺04b] and “full content MathML”. The first subset uses a minimal set of elements representing the meaning of a mathematical expression in a uniform, regular structure, while the second one tries to strike a pragmatic balance between verbosity and formality. The meaning of non-strict expressions is given by a fixed translation: the “strict content MathML translation” specified in section 4.6 of the MathML3 recommendation [ABC⁺10].

This language design has the advantage that only a small, regular sublanguage has to be given a mathematical meaning, but a larger vocabulary that is more intuitive to practitioners of the field can be used for actual representation. Moreover, semantic services like validation only need to be implemented for the strict subset and can be extended to the pragmatic language by translation. Ultimately, a representation format might even have multiple pragmatic front-ends geared towards different audiences. These are semantically interoperable by construction.

The work reported in this paper comes from an ongoing language design effort, where we want to redesign our OMDoc format [Koh06] into a minimal, regular core language (*strict OMDoc 2*) and an extension layer (*pragmatic OMDoc 2*) whose semantics is given by a “pragmatic-to-strict” ($\mathcal{P}2\mathcal{S}$) translation. While this problem is well-understood for mathematical *objects*, extension frameworks *at the statement level* seem to be restricted to the non-semantic case, e.g. the `amsthm` package for $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$.

Languages for mathematics commonly permit a variety of pragmatic statements, e.g., implicit or case-based definitions, type definitions, theorems, or proof schemata. But representation frameworks for such languages do not include a generic mechanism that permits introducing arbitrary pragmatic statements — instead, a fixed set is built into the format. Among logical frameworks, Twelf/LF [PS99,HHP93] permits two statements: defined and undefined constants. Isabelle [Pau94] and Coq [BC04] permit much larger, but still fixed sets that include, for example, recursive case-based function definitions. Content markup formats like OMDoc permit similar fixed sets.

A large set of statements is desirable in a representation format in order to model the flexibility of individual languages. A large *fixed* set on the other hand is unsatisfactory because it is difficult to give a theoretical justification for fixing any specific set of statements. Moreover, it is often difficult to define the semantics of

a built-in statement in a foundationally unconstrained representation format because many pragmatic statements are only meaningful under certain foundational assumptions.

In this paper we present a general formalism for adding new pragmatic statement forms to our OMDoc format; we have picked OMDoc for familiarity and foundation-independence; any other foundational format can be extended similarly. Consider for instance the pragmatic statement of an “implicit definition”, which defines a mathematical object by describing it so accurately, that there is only one object that fits this description. For instance, the exponential function exp is defined as the (unique) solution of the differential equation $f = f'$ with $f(0) = 1$. This form of definition is extensively used in practical mathematics, so pragmatic OMDoc should offer an infrastructure for it, whereas strict OMDoc only offers “simple definitions” of the form $c := d$, where c is a new symbol and d any object. In our extension framework, the $\mathcal{P2S}$ translation provides the semantics of the implicit definition in terms of the strict definition $exp := \iota f.(f' = f \wedge f(0) = 1)$, where ι is a “definite description operator”: Given an expression A with free variable x , such that there is a unique x that makes A valid, $\iota x.A$ returns that x , otherwise $\iota x.A$ is undefined.

Note that the semantics of an implicit definition requires a definite description operator. While most areas of mathematics at least implicitly assume its existence, it should not be required in general because that would prevent the representation of systems without one. Therefore, we make these requirements explicit in a special theory that defines the new pragmatic statement and its strict semantics. This theory must be imported in order for implicit definitions to become available. Using our extension language, we can recover a large number of existing pragmatic statements as definable special cases, including many existing ones of OMDoc. Thus, when representing formal languages in OMDoc, authors have full control what pragmatic statements to permit and can define new ones in terms of existing ones.

In the next section, we will recap those parts of OMDoc that are needed in this paper. In Section 3, we define our extension language, and in Section 4, we look at particular extensions that are motivated by mathematical practice. Finally, in Section 5, we will address the question of extending the concrete syntax with pragmatic features as well.

2 MMT/OMDoc

OMDoc is a comprehensive content-based format for representing mathematical knowledge and documents. It represents mathematical knowledge at three levels: mathematical formulae at the *object level*, symbol declarations, definitions, notation definitions, axioms, theorems, and proofs at the *statement level*, and finally modular scopes at the *theory level*. Moreover, it adds an infrastructure for representing functional aspects of *mathematical documents* at the content markup level. OMDoc

Theories	Documents
Statements	
Objects	

1.2 has been successfully used as a representational basis in applications ranging from theorem prover interfaces, via knowledge based up to eLearning systems. To allow this diversity of applications, the format has acquired a large, interconnected set of language constructs motivated by coverage and user familiarity (i.e., by pragmatic concerns) and not by minimality and orthogonality of language primitives (strict concerns).

To reconcile these language design issues for OMDoc 2, we want to separate the format into a *strict* core language and a *pragmatic* extension layer that is elaborated into strict OMDoc via a “pragmatic-to-strict” ($\mathcal{P2S}$) translation.

For strict OMDoc we employ the foundation-independent, syntactically minimal MMT framework (see below). For pragmatic OMDoc, we aim at a language that is feature-complete with respect to OMDoc 1.2 [Koh06], but incorporates language features from other MKM formats, most notably from Isabelle/Isar [Wen99], PVS [ORS92], and Mizar [TB85].

The MMT language was emerged from a complete redesign of the formal core¹ of OMDoc focusing on foundation-independence, scalability, modularity, while maintaining coverage of formal systems. The MMT language is described in [RK11] and implemented in [Rab08].

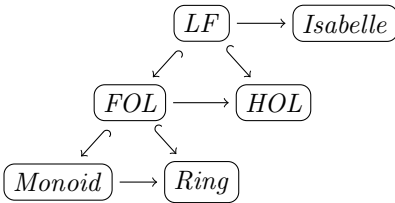


Fig. 1. An MMT Theory Graph

MMT uses **theories** as a single primitive to represent formal systems such as logical frameworks, logics, or theories. These form theory graphs such as the one on the left, where single arrows \rightarrow denote theory translations and hooked arrows \leftrightarrow denote the meta-theory relation between two theories. The theory *FOL* for first-order logic is the meta-theory for *Monoid* and *Ring*. And the theory *LF* for the logical framework LF [HHP93] is the meta-theory of *FOL*

and *HOL* for higher-order logic. In general, we describe the theories with meta-theory *M* as *M-theories*. The importance of meta-theories in MMT is that the syntax and semantics of *M* induces the syntax and semantics of all *M*-theories. For example, if the syntax and semantics are fixed for *LF*, they determine those of *FOL* and *Monoid*.

At the statement level, MMT uses **constant** declarations as a single primitive to represent all OMDoc statement declarations. These are differentiated by the type system of the respective meta-theory. In particular, the Curry-Howard correspondence is used to represent axioms and theorems as plain constants (with special types).

In Figure 2, we show a small fragment of the MMT grammar that we need in the remainder of this paper. Meta-symbols of the BNF format are given in color.

¹ We are currently working on adding an informal (natural language) representation and a non-trivial (strict) document level to MMT, their lack does not restrict the results reported in this paper.

Modules	$G ::= (\mathbf{theory} \ T = \{\Sigma\})^*$
Theories	$\Sigma ::= \cdot \mid \Sigma, c[: E_1][= E_2] \mid \mathbf{meta} \ T$
Contexts	$\Gamma ::= \cdot \mid \Gamma, x[: E]$
Expressions	$E ::= x \mid c \mid E E^+ \mid E \Gamma. E$

Fig. 2. MMT Grammar

The module level of MMT introduces *theory declarations* $\mathbf{theory} \ T = \{\Sigma\}$. Theories Σ contain *constant declarations* $c[: E_1][= E_2]$ that introduce named atomic expressions c with optional type E_1 or definition E_2 . Moreover, each theory may declare its meta-theory T via $\mathbf{meta} \ T$.

MMT expressions are a fragment of OpenMath [\[BCC⁺04a\]](#) objects, for which we introduce a short syntax. They are formed from variables x , constants c , applications $E E_1 \dots E_n$ of functions E to a sequence of arguments E_i , and bindings $E_1 \Gamma. E_2$ that use a binder E_1 , a context Γ of bound variables, and a scope E_2 . Contexts Γ consist of variables $x[: E]$ that can optionally attribute a type E .

The semantics of MMT is given in terms of *foundations* for the upper-most meta-theories. Foundations define in particular the typing relation between expressions, in which MMT is parametric. For example, the foundation for *LF* induces the type-checking relation for all theories with meta-theory *LF*.

Example 1 (MMT-Theories). Below we give an MMT theory *Propositions*, which will serve as the meta-theory of several logics introduced in this paper. It introduces all symbols needed to declare logical connectives and inference rules of a logic. The syntax and semantics of this theory are defined in terms of type theory, e.g., the logical framework LF [\[HHP93\]](#).

type, \rightarrow , and *lam* are untyped constants representing the primitives of type theory. *type* represents the universe of all types, \rightarrow constructs function types $\alpha \rightarrow \beta$, and *lam* represents the λ -binder. *o* is the type of logical formulas and *proof* is a constant that assigns to each logical formula $F : o$ the type *proof* F of its proof.

$$\mathbf{theory} \ Propositions = \{$$

$$\begin{array}{l} \mathbf{type} \\ \rightarrow \\ \mathbf{lam} \\ o : \mathbf{type} \\ \mathbf{proof} : o \rightarrow \mathbf{type} \end{array}$$

$$\}$$

3 A Framework for Language Extensions

We will now define our extension language (EL). It provides a syntactic means to define pragmatic language features and their semantics in terms of strict OMDoc.

Syntax. EL adds two primitive declarations to MMT theories: *extension declarations* and *pragmatic declarations*:

$$\boxed{\begin{array}{l} \Sigma ::= \Sigma, \text{ extension } e = \Phi \\ | \Sigma, \text{ pragmatic } c : \varphi \end{array}}$$

Extension declarations $\text{extension } e = \Phi$ introduce a new declaration schema e that is described by Φ . Intuitively, Φ is a function that takes some arguments and returns a list of declarations, which define the strict semantics of the declaration scheme.

Pragmatic declarations $\text{pragmatic } c : \varphi$ introduce new declarations that make use of a previously declared extension. Intuitively, φ applies an extension e a sequence of arguments and evaluates to the returned list of declarations. Thus, $c : \varphi$ serves as a pragmatic abbreviation of a list of strict declarations.

The key notion in both cases is that of *theory families*. They represent collections of theories by specifying their common syntactic shape. Intuitively, theory families arise by putting a λ -calculus on top of theory fragments Σ :

$$\boxed{\begin{array}{l} \text{Theory Families } \quad \Phi ::= \{\Sigma\} \mid \lambda x : E. \Phi \\ \quad \quad \quad \quad \varphi ::= e \mid \Phi E \end{array}}$$

We group theory families into two non-terminal symbols as shown above: Φ is formed from theory fragments $\{\Sigma\}$ and λ -abstraction $\lambda x : E. \Phi$. And φ is formed from references to previously declared extension e and applications of parametric theory families to arguments E . This has the advantage that both Φ and φ have a very simple shape.

Example 2 (Extension Declarations). In Figure 3 we give the theory *Assertion*, which declares extensions for axiom and theorem declarations. Their semantics is defined in terms of the Curry-Howard representation of strict OMDoc.

Both extensions take a logical formula $F : o$ as a parameter. The extension *axiom* permits pragmatic declarations of the form $c : \text{axiom } F$. These abbreviate MMT constant declarations of the form $c : \text{proof } F$.

The extension *theorem* additionally takes a parameter $D : \text{proof } F$, which is a proof of F . It permits pragmatic declarations of the form $c : \text{theorem } F D$. These abbreviate MMT constant declarations of the form $c : \text{proof } F = D$.

```

theory Assertion = {
  meta Propositions
  extension axiom =  $\lambda F : o. \{$ 
     $c : \text{proof } F$ 
   $\}$ 
  extension theorem =  $\lambda F : o. \lambda D : \text{proof } F. \{$ 
     $c : \text{proof } F = D$ 
   $\}$ 
}

```

Fig. 3. An MMT Theory with Extension Declarations

Any MMT theory may introduce extension declarations. However, pragmatic declarations are only legal if the extension that is used has been declared in the meta-theory:

Definition 1 (Legal Extension Declarations). *We say that an extension declaration $\text{extension } e = \lambda x_1 : E_1. \dots \lambda x_n : E_n. \{\Sigma\}$ is **legal** in an MMT theory T , if the declarations $x_1 : E_1, \dots, x_n : E_n$ and Σ are well-formed in T .*

This includes the case where Σ contains pragmatic declarations.

Definition 2 (Legal Pragmatic Declarations). *We say that a pragmatic declaration $\text{pragmatic } c : e E_1 \dots E_n$ is **legal** in an MMT theory T if there is a declaration extension $e = \lambda x_1 : E'_1. \dots \lambda x_n : E'_n. \{\Sigma\}$ in the meta-theory of T and each E_i has type E'_i .*

Here the typing relation is the one provided by the MMT foundation.

Semantics. Extension declarations do not have a semantics as such because the extension declared in M only govern what pragmatic declarations are legal in M -theories. In particular, contrary to the constant declarations in M , a model of M does not interpret the extension declarations.

The semantics of pragmatic declarations is given by elaborating them into strict declarations:

Definition 3 (Pragmatic-to-Strict Translation $\mathcal{P2S}$). *A legal pragmatic declaration $\text{pragmatic } c : e E_1 \dots E_n$ is translated to a list of strict constant declarations*

$$c.d_1 : \gamma(F_1) = \gamma(D_1), \dots, c.d_m : \gamma(F_m) = \gamma(D_m)$$

where γ substitutes every x_i with E_i and every d_j with $c.d_j$ if we have

$$\text{extension } e = \lambda x_1 : E'_1. \dots \lambda x_n : E'_n. \{d_1 : F_1 = D_1, \dots, d_m : F_m = D_m\}$$

and every expression E_i has type E'_i .

Example 3. Consider the following MMT theories in Figure 4: *HOL* includes the MMT theory *Propositions* and declares a constant i as the type of individuals. It adds the usual logical connectives and quantifiers – here we only present truth (*true*) and the universal quantifier (\forall) – and introduces equality (\doteq) on expressions of type α . Then it includes *Assertion*. This gives *HOL* access to the extensions *axiom* and *theorem*.

Commutativity uses *HOL* as its meta-theory and declares a constant \circ that takes two individuals as arguments and returns an individual. It adds a pragmatic declaration named *comm* that declares the commutativity axiom for \circ using the axiom extension from *HOL*.

Commutativity' is obtained by elaborating *Commutativity* according to Definition 3.

<pre> theory HOL = { meta Propositions i : type true : o : ∀ : (α → o) → o ≐ : α → α → o include Assertion } </pre>	<pre> theory Commutativity = { meta HOL o : i → i → i pragmatic comm : axiom ∀x : i. ∀y : i. x o y ≐ y o x } theory Commutativity' = { meta HOL o : i → i → i comm.c : proof ∀x : i. ∀y : i. x o y ≐ y o x } </pre>
---	--

Fig. 4. A $\mathcal{P}2\mathcal{S}$ Translation Example

4 Representing Extension Principles

Formal mathematical developments can be classified based on whether they follow the axiomatic or the definitional method. The former is common for logics where theories declare primitive constants and axioms. The latter is common for foundations of mathematics where a fixed theory (the foundation) is extended only by defined constants and theorems. In MMT, both the logic and the foundation are represented as a meta-theory M , and the main difference is that the definitional method does not permit undefined constants in M -theories.

However, this treatment does not capture conservative extension principles: These are meta-theorems that establish that certain extensions are acceptable even if they are not definitional. We can understand them as intermediates between axiomatic and definitional extensions: They may be axiomatic but are essentially as safe as definitional ones.

To make this argument precise, we use the following definition:

Definition 4. We call the theory family $\Phi = \lambda x_1 : E'_1. \dots \lambda x_n : E'_n. \{\Sigma\}$ **conservative** for M if for every M -theory T and all $E_1 : E'_1, \dots, E_n : E'_n$, every model of T can be extended to a model of $T, \gamma(\Sigma)$, where γ substitutes every x_i with E_i .

An extension declaration $\text{extension } e = \Phi$ is called **derived** if all constant declarations in Σ have a definiens; otherwise, it is called **primitive**.

Primitive extension declarations correspond to axiom declarations because they postulate that certain extensions of M are legal. The proof that they are indeed conservative is a meta-argument that must be carried out as a part of the proof that M is an adequate MMT representation of the represented formalism. Similarly, derived extension declarations correspond to theorem declarations because their conservativity follows from that of the primitive ones. More precisely: If all primitive extension principles in M are conservative, then so are all derived ones.

In the following, we will recover built-in extension statements of common representation formats as special cases of our extension declarations. We will follow a *little foundations* paradigm and state every extensions in the smallest theory in which it is meaningful. Using the MMT module system, this permits maximal reuse of extension definitions. Moreover, it documents the (often implicit) foundational assumptions of each extension.

Implicit Definitions in OMDoc. Implicit definitions of OMDoc 1.2 are captured using the following derived extension declaration. If the theory *ImplicitDefinitions* in Figure 5 is included into a meta-theory M , then M -theories may use implicit definitions.

```

theory ImplicitDefinitions = {
  meta Propositions
   $\exists^!$  :  $(\alpha \rightarrow o) \rightarrow o$ 
   $\iota$  :  $(\alpha \rightarrow o) \rightarrow \alpha$ 
   $\iota_{ax}$  : proof  $\exists^! x P x \rightarrow$  proof  $P (\iota P)$ 
  extension impldef =  $\lambda \alpha : \mathbf{type}. \lambda P : \alpha \rightarrow o. \lambda m : \mathbf{proof} \exists^! x : \alpha. P x. \{$ 
     $c$  :  $\alpha = \iota P$ 
     $c_{ax}$  : proof  $\exists^! x : \alpha. P x$ 
  }
}
    
```

Fig. 5. An Extension for Implicit Definitions

Note that *ImplicitDefinitions* requires two other connectives: A description operator (ι) and a unique existential ($\exists^!$) are needed to express the meaning of an implicit definition. We deliberately assume only those two operators in order to maximize the re-usability of this theory: Using the MMT module system, any logic M in which these two operators are definable can import the theory *ImplicitDefinitions*.

More specifically, *ImplicitDefinitions* introduces the definite description operator as a new binding operator (ι), and describes its meaning by the axiom $\exists^! x P(x) \Rightarrow P(\iota P)$ formulated in ι_{ax} for any predicate P on α . The extension *impldef* permits pragmatic declarations of the form $f : \mathit{impldef} \alpha P m$, which defines f as the unique object which makes the property P valid. This leads to the well-defined condition that there is indeed such a unique object, which is discharged by the proof m . The pragmatic-to-strict translation from Section 3 translates the pragmatic declaration $f : \mathit{impldef} \alpha P m$ to the strict constant declarations $f.c : \alpha = \iota P$ and $f.c_{ax} : \mathit{proof} \exists^! x : \alpha P x$.

Mizar-Style Functor Definitions. The Mizar language [TB85] provides a wide (but fixed) variety of special statements, most of which can be understood as conservative extension principles for first-order logic. A comprehensive list of the corresponding extension declarations can be found in [IKR11]. We will only consider one example in Figure 6.

```

theory FunctorDefinitions = {
  meta Propositions
   $\wedge : o \rightarrow o \rightarrow o$ 
   $\Rightarrow : o \rightarrow o \rightarrow o$ 
   $\forall : (\alpha \rightarrow o) \rightarrow \alpha$ 
   $\exists : (\alpha \rightarrow o) \rightarrow \alpha$ 
   $\doteq : \alpha \rightarrow \alpha \rightarrow o$ 
  extension functor =
     $\lambda\alpha : \mathbf{type}. \lambda\beta : \mathbf{type}. \lambda\mathit{means} : \alpha \rightarrow \beta \rightarrow o.$ 
     $\lambda\mathit{existence} : \mathit{proof} \forall x : \alpha. \exists y : \beta. \mathit{means} \ x \ y.$ 
     $\lambda\mathit{uniqueness} : \mathit{proof} \forall x : \alpha. \forall y : \beta. \forall y' : \beta. \mathit{means} \ x \ y \wedge \mathit{means} \ x \ y' \Rightarrow y \doteq y'. \{$ 
       $f$ 
       $: \alpha \rightarrow \beta$ 
       $\mathit{definitional\_theorem} : \mathit{proof} \forall x : \alpha. \mathit{means} \ x \ (f \ x)$ 
     $\}$ 
  }
}

```

Fig. 6. An Extension for Mizar-Style Functor Definitions

The theory *FunctorDefinitions* describes Mizar-style implicit definition of a unary function symbol (called a *functor* in Mizar). This is different from the one above because it uses a primitive extension declaration that is well-known to be conservative. In Mizar, the axiom *definitional_theorem* is called the definitional theorem induced by the implicit definition. Using the extension *functor*, one can introduce pragmatic declarations of the form **pragmatic** $c : \mathit{functor} \ A \ B \ P \ E \ U$ that declare functors c from A to B that are defined by the property P where E and U discharge the induced proof obligations.

Flexary Extensions. The above two examples become substantially more powerful if they are extended to implicit definitions of functions of arbitrary arity. This is supported by our extension language by using an LF-based logical framework with term sequences and type sequences. We omit the formal details of this framework here for simplicity and refer to [Hor12] instead. We only give one example in Figure 7 that demonstrates the potential.

```

theory CaseBasedDefinitions = {
  meta Propositions
   $\wedge : o^n \rightarrow o$ 
   $\vee^! : o^n \rightarrow o$ 
   $\Rightarrow : o \rightarrow o \rightarrow o$ 
   $\forall : (\alpha \rightarrow o) \rightarrow o$ 
  extension casedef =  $\lambda n : \mathbb{N}. \lambda\alpha : \mathbf{type}. \lambda\beta : \mathbf{type}. \lambda c : (\alpha \rightarrow o)^n.$ 
     $\lambda d : (\alpha \rightarrow \beta)^n. \lambda\rho : \mathit{proof} \forall x : \alpha. \vee^! [c_i \ x]_{i=1}^n. \{$ 
       $f : \alpha \rightarrow \beta$ 
       $a x : \mathit{proof} \forall x : \alpha. \wedge [c_i \ x \Rightarrow (f \ x) = (d_i \ x)]_{i=1}^n$ 
     $\}$ 
  }
}

```

Fig. 7. An Extension for Case-Based Definitions

The theory *CaseBasedDefinitions* introduces an extension that describes the case-based definition of a unary function f from α to β that is defined using n different cases where each case is guarded by the predicate c_i together with the respective definiens d_i . Such a definition is well-defined if for all $x \in \alpha$ exactly one out of the $c_i x$ is true. Note that these declarations use a special sequence constructor: for example, $[c_i x]_{i=1}^n$ simplifies to the sequence $c_1 x, \dots, c_n x$. Moreover, \wedge and $\vee^!$ are flexary connectives, i.e., they take a flexible number of arguments. In particular, $\vee^!(F_1, \dots, F_n)$ holds if exactly one of its arguments holds.

The pragmatic declaration **pragmatic** $f : \text{casedef } n \alpha \beta c_1 \dots c_n d_1 \dots d_n \rho$ corresponds to the following function definition:

$$f(x) = \begin{cases} d_1(x) & \text{if } c_1(x) \\ \vdots & \vdots \\ d_n(x) & \text{if } c_n(x) \end{cases}$$

HOL-Style Type Definitions. Due to the presence of λ -abstraction and a description operator in HOL [Chu40], a lot of common extension principles become derivable in HOL, in particular, implicit definitions.

But there is one primitive definition principle that is commonly accepted in HOL-based formalizations of the definitional method: A Gordon/HOL type definition [Gor88] introduces a new type that is axiomatized to be isomorphic to a subtype of an existing type. This cannot be expressed as a derivable extension because HOL does not use subtyping.

```

theory Types = {
  meta Propositions
   $\forall : (\alpha \rightarrow o) \rightarrow o$ 
   $\exists : (\alpha \rightarrow o) \rightarrow o$ 
   $\doteq : (\alpha \rightarrow \alpha) \rightarrow o$ 
  extension typedef =  $\lambda \alpha : \text{type}. \lambda A : \alpha \rightarrow o. \lambda P : \text{proof } \exists x : \alpha. A x. \{$ 
    T      : type
    Rep    :  $T \rightarrow \alpha$ 
    Abs    :  $\alpha \rightarrow T$ 
    Rep'   :  $\text{proof } \forall x : T. A (\text{Rep } x)$ 
    Rep_inverse :  $\text{proof } \forall x : T. \text{Abs } (\text{Rep } x) \doteq x$ 
    Abs_inverse :  $\text{proof } \forall x : \alpha. A x \Rightarrow \text{Rep } (\text{Abs } x) \doteq x$ 
  }
}
    
```

Fig. 8. An Extension for HOL-Style Type Definitions

The theory *Types* in Figure 8, formalizes this extension principle. Our symbol names follow the implementation of this definition principle in Isabelle/HOL [NPW02]. Pragmatic declarations of the form **pragmatic** $t : \text{typedef } \alpha A P$ introduce a new non-empty type t isomorphic to the predicate A over α . Since all HOL-types must be non-empty, a proof P of the non-emptiness of A must be supplied. More precisely, it is translated to the following strict constant declarations:

- $t.T : \text{type}$ is the new type that is being defined,
- $t.Rep : t.T \rightarrow \alpha$ is an injection from the new type $t.T$ to α ,
- $t.Abs : \alpha \rightarrow t.T$ is the inverse of $t.Rep$ from α to the new type $t.T$,
- $t.Rep'$ states that the property A holds for any term of type $t.T$,
- $t.Rep_inverse$ states that the injection of any element of type $t.T$ to α and back is equal to itself,
- $t.Abs_inverse$ states that if an element satisfies A , then injecting it to $t.T$ and back is equal to itself.

HOL-based proof assistants implement the type definition principle as a built-in statement. They also often provide further built-in statements for other definition principles that become derivable in the presence of type definitions, e.g., a definition principle for record types. For example, in Isabelle/HOL [NPW02], HOL is formalized in the Pure logic underlying the logical framework Isabelle [Pau94]. But because the type definition principle is not expressible in Pure, it is implemented as a primitive Isabelle feature that is only active in Isabelle/HOL.

5 Syntax Extensions and Surface Languages

Our definitions from Section 3 permit pragmatic *abstract* syntax, which is elaborated into strict abstract syntax. For human-oriented representations, it is desirable to complement this with similar extensions of pragmatic *concrete* syntax. While the pragmatic-to-strict translation at the abstract syntax level is usually non-trivial and therefore not invertible, the corresponding translation at the concrete syntax level should be compositional and bidirectional.

5.1 OMDoc Concrete Syntax for EL Declarations

First we extend OMDoc with concrete syntax that exactly mirrors the abstract syntax from Section 3. The declaration **extension** $e = \lambda x_1 : E_1 \dots \lambda x_n : E_n. \{\Sigma\}$ is written as

```
<extension name="e">
  <parameter name="x1">E1</parameter>
  ⋮
  <parameter name="xn">En</parameter>
  <theory>
    Σ
  </theory>
</extension>
```

Here we use the box notation \boxed{A} to gloss the XML representation of an entity A given in abstract syntax.

Similarly, the pragmatic declaration **pragmatic** $c : e E_1 \dots E_n$ is written as


```
<pragmatic name="c" extension="⟨M⟩?e">
   $E_1 \dots E_n$ 
</pragmatic>
```

Here $\langle M \rangle$ is the meta-theory in which e is declared so that $\langle M \rangle?e$ is the MMT URI of the extension.

Example 4. For the implicit definitions discussed in Section 3, we use the extension *impldef* from Figure 5, which we assume has namespace URI $\langle U \rangle$. If ρ is a proof of unique existence for an f such that $f' = f \wedge f(0) = 1$, then the exponential function is defined in XML by

```
<pragmatic name="exp" extension="⟨U⟩?ImplicitDefinitions?impldef">
   $\lambda f.f' = f \wedge f(0) = 1$   $\rho$ 
</pragmatic>
```

5.2 Pragmatic Surface Syntax

OMDoc is mainly a machine-oriented interoperability format, which is not intended for human consumption. Therefore, the EL-isomorphic syntax introduced is sufficient in principle – at least for the formal subset of OMDoc we have discussed so far.

OMDoc is largely written in the form of “surface languages” – domain-specific languages that can be written effectively and transformed to OMDoc in an automated process. For the formal subset of OMDoc, we use a MMT-inspired superset of the Twelf/LF [PS99,HHP93] syntax, and for informal OMDoc we use $\mathcal{S}\text{T}_{\text{E}}\text{X}$ [Koh08], a semantic extension of $\text{T}_{\text{E}}\text{X}/\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$.

For many purposes like learning the surface language or styling OMDoc documents, **pragmatic surface syntax**, i.e., a surface syntax that is closer to the notational conventions of the respective domain, has great practical advantages. It is possible to support, i.e., generate and parse, pragmatic surface syntax by using the macro/scripting framework associated with most representation formats.

For instance, we can regain the XML syntax familiar from OMDoc 1.2 via notation definitions that transform between **pragmatic** elements and the corresponding OMDoc 1.2 syntax. For Twelf/LF, we would extend the module system preprocessor, and for Isabelle we would extend the SML-based syntax/parsing subsystem. We have also extended $\mathcal{S}\text{T}_{\text{E}}\text{X}$ as an example of a semi-formal surface language. Here we used the macro facility of $\text{T}_{\text{E}}\text{X}$ as the computational engine. We conjecture that most practical surface languages for MKM can be extended similarly.

These translations proceed in two steps. Firstly, pragmatic surface syntax is translated into our pragmatic MMT syntax. Our language is designed to make this step trivial: in particular, it does not have to look into the parameters used in a pragmatic surface declaration. Secondly, pragmatic MMT syntax is

type-checked and, if desired, translated into strict MMT syntax. All potentially difficult semantic analysis is part of this second step. This design makes it very easy for users to introduce their own pragmatic surface syntax.

6 Conclusion and Future Work

In this paper, we proposed a general statement-level extension mechanism for MKM formats powered by the notion of theory families. Starting with MMT as a core language, we are able to express most of the pragmatic language features of OMDoc 1.2 as instances of our new extension primitive. Moreover, we can recover extension principles employed in languages for formalized mathematics including the statements employed for conservative extensions in Isabelle/HOL and Mizar. We have also described a principle how to introduce corresponding pragmatic concrete syntax.

The elegance and utility of the extension language is enhanced by the modularity of the OMDoc 2 framework, whose meta-theories provide the natural place to declare extensions: the scoping rules of the MMT module system supply the justification and intended visibility of statement-level extensions. In our examples, the Isabelle/HOL and Mizar extensions come from their meta-logics, which are formalized in MMT.

We also expect our pragmatic syntax to be beneficial in system integration because it permit interchanging documents at the pragmatic MMT level. For example, we can translate implicit definitions of one system to those of another system even if – as is typical – the respective strict implementations are very different.

For full coverage of OMDoc 1.2, we still need to capture abstract data types and proofs; the difficulties in this endeavor lie not in the extension framework but in the design of suitable meta-logics that justify them. For OMDoc-style proofs, the $\bar{\lambda}\mu\tilde{\mu}$ -calculus has been identified as suitable [ASC06], but remains to be encoded in MMT. For abstract data types we need a λ -calculus that can reflect signatures into (inductive) data types; the third author is currently working on this.

The fact that pragmatic extensions are declared in meta-theories points towards the idea that OMDoc metadata and the corresponding metadata ontologies [LK09] are actually meta-theories as well (albeit at a somewhat different level); we plan to work out this correspondence for OMDoc 2.

Finally, we observe that we can go even further and interpret the feature of definitions that is primitive in MMT as pragmatic extensions of an even more foundational system. Then definitions $c : E = E'$ become pragmatic notations for a declaration $c : E$ and an axiom $c = E'$, where $=$ is an extension symbol introduced in a meta-theory for equality. Typing can be handled similarly. This would also permit introducing other modifiers in declarations such as $<$: for subtype declarations.

References

- ABC⁺10. Ausbrooks, R., Buswell, S., Carlisle, D., Chavchanidze, G., Dalmas, S., Devitt, S., Diaz, A., Dooley, S., Hunter, R., Ion, P., Kohlhase, M., Lazrek, A., Libbrecht, P., Miller, B., Miner, R., Sargent, M., Smith, B., Soiffer, N., Sutor, R., Watt, S.: Mathematical Markup Language (MathML) version 3.0. W3C Recommendation, World Wide Web Consortium (W3C) (2010)
- ASC06. Autexier, S., Sacerdoti-Coen, C.: A Formal Correspondence Between OMDoc with Alternative Proofs and the $\bar{\lambda}\mu\tilde{\mu}$ -Calculus. In: Borwein, J.M., Farmer, W.M. (eds.) MKM 2006. LNCS (LNAI), vol. 4108, pp. 67–81. Springer, Heidelberg (2006)
- BC04. Bertot, Y., Castéran, P.: Coq'Art: The Calculus of Inductive Constructions. Springer (2004)
- BCC⁺04a. Buswell, S., Caprotti, O., Carlisle, D., Dewar, M., Gaetano, M., Kohlhase, M.: The Open Math Standard, Version 2.0. Technical report, The Open Math Society (2004), <http://www.openmath.org/standard/om20>
- BCC⁺04b. Buswell, S., Caprotti, O., Carlisle, D.P., Dewar, M.C., Gaetano, M., Kohlhase, M.: The Open Math standard, version 2.0. Technical report, The OpenMath Society (2004)
- Chu40. Church, A.: A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic* 5(1), 56–68 (1940)
- Gor88. Gordon, M.: HOL: A Proof Generating System for Higher-Order Logic. In: Birtwistle, G., Subrahmanyam, P. (eds.) VLSI Specification, Verification and Synthesis, pp. 73–128. Kluwer-Academic Publishers (1988)
- HHP93. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. *Journal of the Association for Computing Machinery* 40(1), 143–184 (1993)
- Hor12. Horozal, F.: Logic translations with declaration patterns (2012), <https://svn.kwarc.info/repos/fhorozal/pubs/patterns.pdf>
- IKR11. Iancu, M., Kohlhase, M., Rabe, F.: Translating the Mizar Mathematical Library into OMDoc format. Technical Report KWARC Report-01/11, Jacobs University Bremen (2011)
- Koh06. Kohlhase, M.: OMDoc – An Open Markup Format for Mathematical Documents [version 1.2]. LNCS (LNAI), vol. 4180. Springer, Heidelberg (2006)
- Koh08. Kohlhase, M.: Using $\mathcal{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ as a semantic markup format. *Mathematics in Computer Science* 2(2), 279–304 (2008)
- LK09. Lange, C., Kohlhase, M.: A Mathematical Approach to Ontology Authoring and Documentation. In: Carette, J., Dixon, L., Coen, C.S., Watt, S.M. (eds.) MKM 2009, Held as Part of CICM 2009. LNCS, vol. 5625, pp. 389–404. Springer, Heidelberg (2009)
- NPW02. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Springer (2002)
- ORS92. Owre, S., Rushby, J., Shankar, N.: PVS: A Prototype Verification System. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992)
- Pau94. Paulson, L.: Isabelle: A Generic Theorem Prover. LNCS, vol. 828. Springer, Heidelberg (1994)
- PS99. Pfenning, F., Schürmann, C.: System Description: Twelf - A Meta-Logical Framework for Deductive Systems. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 202–206. Springer, Heidelberg (1999)
- Rab08. Rabe, F.: The MMT System (2008), <https://trac.kwarc.info/MMT/>

- RK11. Rabe, F., Kohlhase, M.: A Scalable Module System (2011), <http://arxiv.org/abs/1105.0548>
- TB85. Trybulec, A., Blair, H.: Computer Assisted Reasoning with MIZAR. In: Joshi, A. (ed.) Proceedings of the 9th International Joint Conference on Artificial Intelligence, pp. 26–28 (1985)
- Wen99. Wenzel, M.T.: Isar - A Generic Interpretative Approach to Readable Formal Proof Documents. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) TPHOLs 1999. LNCS, vol. 1690, pp. 167–184. Springer, Heidelberg (1999)

A Streaming Digital Ink Framework for Multi-party Collaboration

Rui Hu, Vadim Mazalov, and Stephen M. Watt

The University of Western Ontario
London Ontario, Canada N6A 5B7
{rhu8, vmazalov, Stephen.Watt}@uwo.ca

Abstract. We present a framework for pen-based, multi-user, online collaboration in mathematical domains. This environment provides participants, who may be in the same room or across the planet, with a shared whiteboard and voice channel. The digital ink stream is transmitted as InkML, allowing special recognizers for different content types, such as mathematics and diagrams. Sessions may be recorded and stored for later playback, analysis or annotation. The framework is currently structured to use the popular Skype and Google Talk services for the communications channel, but other transport mechanisms could be used. The goal of the work is to support computer-enhanced distance collaboration, where domain-specific recognizers handle different kinds of digital ink input and editing. The first of these recognizers is for mathematics, which allows converting math input into machine-understandable format. This supports multi-party collaboration, with sessions recorded in rich formats that allow semantic analysis and manipulation of the content.

Keywords: Pen computing, distance collaboration, mathematical handwriting recognition, InkML, Skype, Google Talk.

1 Introduction

We are interested in development of an infrastructure that helps researchers, engineers, teachers and students to collaborate online over pen-based and graphical interfaces. Pen-based collaboration in mathematical domains can significantly increase productivity, e.g. Gowers, et al. conducted an experiment of “attacking” a mathematical problem through a collaboration of volunteers online [1]. In just over five weeks, more than two dozen individuals contributed approximately 800 comments that led to the solution of the problem. We believe that the synergy of pen-based *collaboration* and *recognition* of mathematical input can enhance the efficiency of online interaction. Nevertheless, there is no technology that allows to capitalize on both simultaneously: some software handles recognition without the ability for real-time sharing, e.g. the Maple computer algebra system [2], while other systems provide a whiteboard for collaboration, but no mathematical recognition, e.g. Microsoft OneNote [3], Calliflower [4] or Dabbleboard [5].

We present a framework for multi-user online collaboration in a pen-based and graphical environment. This environment allows participants conducting

and archiving collaborative sessions that involve synchronized voice and digital ink on a shared canvas. The digital ink is represented as InkML [6], allowing special recognizers for different content types, such as mathematics and diagrams. The collaborative sessions may be recorded and stored for later playback, analysis or annotation. The framework currently employs the popular Skype [7] or Google Talk [8] services as the backbone to deliver data streams, but other transport mechanisms could be used. Our objective is to support computer-enhanced distance collaboration, where domain-specific recognizers handle different kinds of digital ink input and editing. The first of these domains is mathematical input, which has similarities to both natural language handwriting input and two-dimensional diagramming. This framework has potential to increase productivity in teleconferences or to enhance online learning and tutoring, as the participants can interact in a natural way. A version of the framework implementation is available for download at <http://www.orcca.on.ca/InkChat/>. The current release (version 0.9.5) has many of the features described in the paper. Some of the features described here have been implemented in separate packages and will be integrated in later versions of InkChat. This work is an outgrowth of earlier work [9] that allowed collaborative inking, but which did not allow modular extension.

The remainder of this article is organized as follows. In Section 2, we explain why it is important for collaboration software to be portable. We then investigate potential challenges that may be encountered in the development and propose our solutions. Section 3 presents a high-level overview of the architecture of the framework and gives details on each component. In Section 4, we describe the implementation of the framework. A case study and a discussion of lessons learnt are given in Section 5. Section 6 concludes the article.

2 Portability of the Framework

A number of digital ink applications have been developed over the past years, following the emergence of digital ink. Most of these applications lack support for portability and are consequently each restricted to a single platform. Collaboration software, however, is most useful when it is platform-independent, both because an individual may use different platforms at different times, and because teams may be composed of members using different systems. Dealing with significantly different client software reduces the ability of the individual or the group to master its use. If any of the members has difficulty, efficiency of the whole team is reduced. Those tools that are cross-platform usually can work only with proprietary ink standards and thus are restricted in their ability to share data with other applications. In this section we investigate the challenges of building a portable framework.

2.1 Portability of Software

Today's platforms capture digital ink in various data structures and process it using different mechanisms. Our framework must therefore deal with different

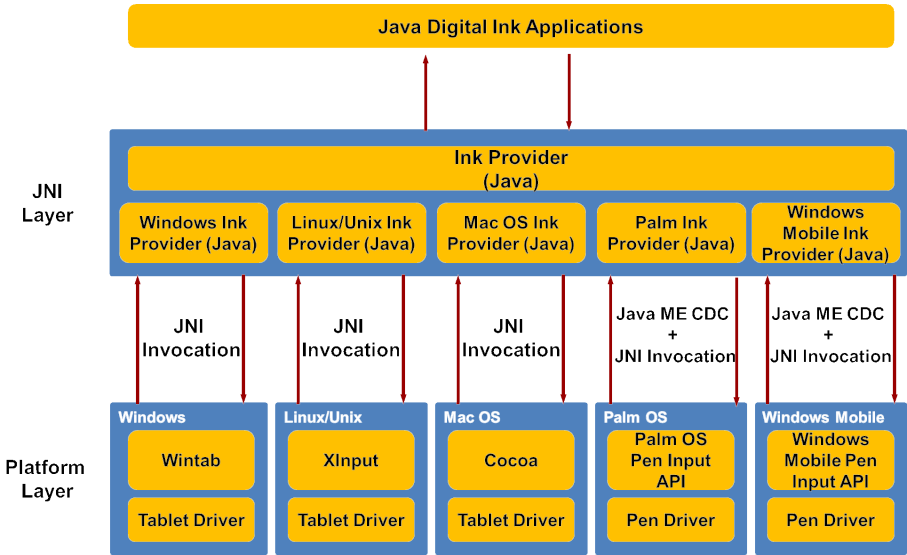


Fig. 1. A cross-platform framework for digital ink applications

lower-level interfaces on different platforms. On Windows, digital ink is captured in Wintab packets or `Stroke` objects and transmitted by the Wintab interface or the Windows XP Tablet PC APIs. A similar scheme is applicable to Linux platforms that use `XInput` events and the Linux Input Subsystem. For Mac OS X we use the `NSEvent` objects and the Cocoa Framework. These platform-specific APIs make development of pen-based collaboration software difficult.

Having explored available APIs on a variety of platforms, including Windows, Windows Mobile, Linux, Mac OS X and Palm OS, we use a framework that can capture digital ink across these platforms and provide a platform-independent, consistent interface to digital ink applications. This framework, as illustrated in Figure 1, contains two layers: the platform layer and the Java Native Interface (JNI) layer. The platform layer receives digital ink input from drivers and passes the data to the upper interfaces: Wintab for Windows, XInput for Linux/Unix and Cocoa for Mac OS. These interfaces push the data to the user space in a platform-specific way and describe each data event inconsistently. This puts the responsibility for event conformance on the shoulders of developers of ink applications. One of our objectives is to allow developers to focus on functionality, rather than the platform-specific issues. This leads to the design of the JNI layer.

The JNI layer interacts with the platform layer and provides consistent, platform-independent APIs for digital ink applications. As the input APIs are implemented in C-like languages, the JNI layer can invoke them using the Java Native Interface. The JNI layer collects digital ink input from the platform layer, converts it to platform-independent events and then dispatches to digital ink applications. This allows development of pen-based applications on top of the JNI layer to be platform-independent.

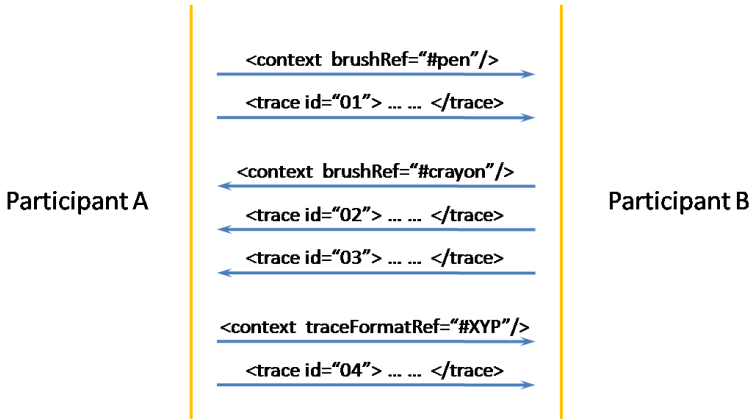


Fig. 2. Ink streaming

2.2 Portability of Digital Ink Data

Another challenge in developing whiteboard-diagramming software is associated with the heterogeneous environment. Various pen devices have different characteristics, including sampling rate, channel properties, screen resolution and so on. The representation of the data generated by a device affects the usability of an application. Previous whiteboard tools did not address this as thoroughly as we require. An example is InkBoard [10], a collaborative sketching application based on Microsoft’s Conference XP research platform [11] and designed for Tablet PCs. InkBoard allows design teams to interact by streaming digital ink in Microsoft’s proprietary Ink Serialized Format (ISF) [12]. As a result, the application is limited in use to Windows environments where ISF is natively supported.

To represent digital ink, we find it useful to adopt an open, flexible, powerful, platform-, and vendor-independent standard, such as InkML [6]. This allows complete and accurate representation of digital ink by capturing the recording information such as the device characteristics, pen tilt, pen pressure and so on. Most importantly, it provides support for collaboration applications that require streaming ink between participants.

The InkML streaming of digital ink is based on the concept of “context.” Whenever digital ink is written, there is some context in effect. Contexts may be represented externally using the `<context>` element in InkML. This can contain various associated attributes, including canvas properties, canvas transformation, trace format, ink source metadata, and time stamp. Initially, each ink collaboration participant obtains a default context and listens for context changes. This is similar to an event-driven model in which context changes are made when contextual elements are received. In practice, these elements will interperse among digital ink streams, as shown in Figure 2. With this model, each participant can easily maintain the current context of the sender in addition to

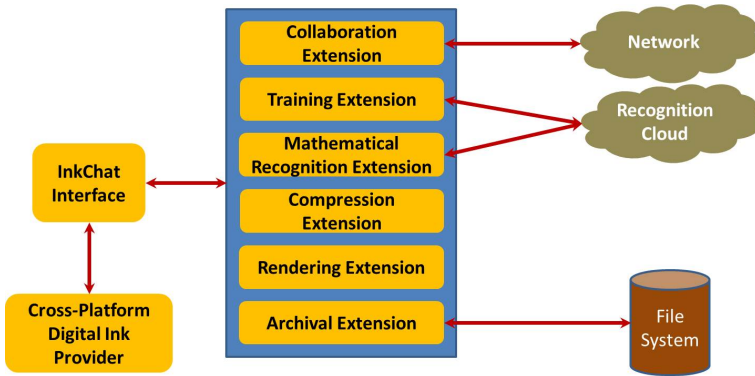


Fig. 3. InkChat architecture

its own local context. Whenever a new contextual element is received, it simply updates old values. Contextual elements are sent only when there is a context change, which helps to decrease streaming overhead on the wire. The overhead can be further reduced by making references to existing contextual elements, which can be pre-defined or previously received. This can be accomplished by using referencing attributes (e.g. `brushRef`) of `<context>`.

3 Architecture

We now present the framework for multi-user online collaboration, allowing sessions to be recorded in formats that allow semantic analysis and manipulation of the content. The framework currently consists of InkChat, a digital ink application developed at Ontario Research Centre for Computer Algebra, and a number of extensions. InkChat is the main platform of the application, on top of which other extensions may be added. Figure 3 presents a high-level overview of the architecture of InkChat. InkChat interacts with the cross-platform framework, presented in Section 2.1, whose primary purpose is to collect digital ink from a variety of platforms and to provide a platform-independent, consistent interface for digital ink applications. The six extensions, most of which can work independently and simultaneously, serve as plug-ins for InkChat. We outline the details of each extension below.

3.1 The Collaboration Extension

Similar to the traditional concept of sharing a session between several parties available in communication software (e.g. having a group call or a text chat), the collaboration extension allows real-time sharing of the canvas among participants and enables the participants to edit content on the canvas. Synchronization of the canvas is performed through the underlying communication backbone.

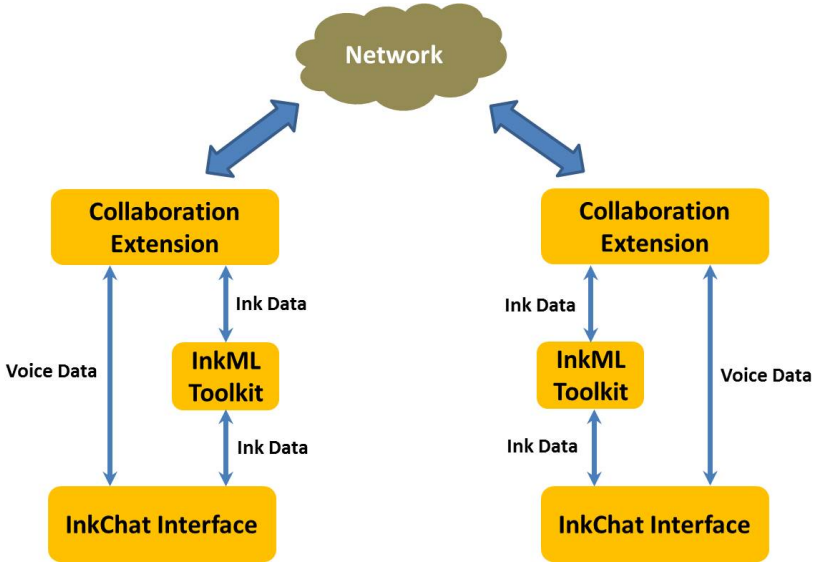


Fig. 4. Collaboration framework

In addition, the voice and video channels are typically available through the communication software as well. The collaboration scheme between two clients is shown in Figure 4. Digital ink data is first converted into InkML format and then sent in a data channel parallel to the voice channel. The collaboration extension itself is an abstract layer that can handle different network protocols including pipes and sockets in P2P (peer-to-peer) and the traditional client-server configurations. To adapt to the collaboration environment, we allow participants to choose the most suitable protocol at the beginning of the conversation. For example, if the collaboration is intended to take place in a small group and requires only the basic functions of whiteboard, P2P would be preferable as it is inexpensive and fairly simple to set up and manage. If the collaboration is intended to comprise a large number of participants and demand sophisticated computing services (e.g. mathematical formula simplification), client-server mode may be more appropriate.

InkChat supports conference mode where more than two participants can be involved in one conversation. Depending on the chosen network protocol, InkChat employs different mechanism to communicate with other participants. When a P2P backbone such as Skype is used, the conference is initiated by the host that has a connection with every other participant. Digital ink routing shares the same mechanism as audio routing, each ink stroke will be broadcast by the host to all participants except the initiator. Figure 5 shows the two cases, when a host and when a client initiate a stroke. In the client-server network, the server will play the role of the host and establish a connection with each client, as shown in Figure 6. Both the digital ink and audio data are sent to the server first and then broadcast to all the other clients.

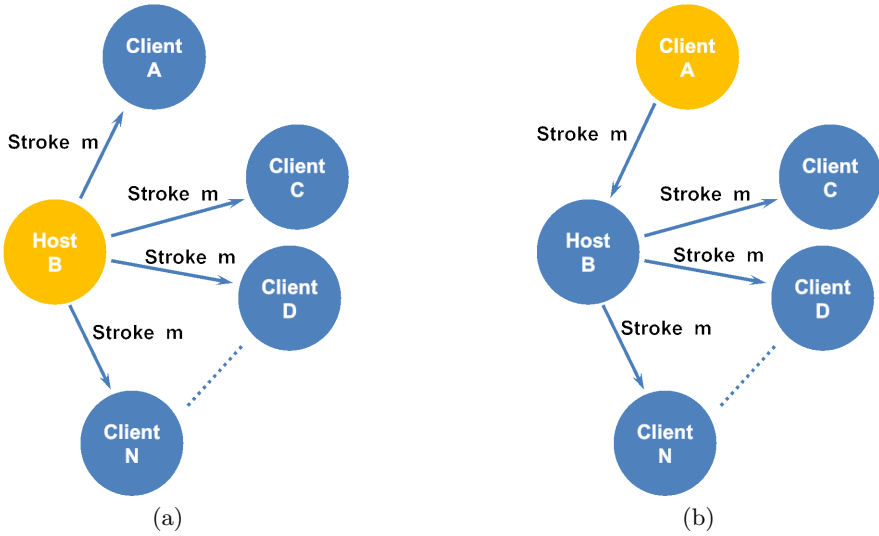


Fig. 5. Sessions with the stroke initiated by (a) the host and (b) the client

3.2 The Training Extension

In an adaptive recognition environment, a separate training phase is not required. Having some number of training samples in each class can, however, significantly improve the initial recognition. The number of training samples that a class should contain depends on the recognition methods. Using our approach, the recognition rate depends on the minimal distance to convex hulls of neighbouring classes. For most of the classes in our dataset, about 20 classes are required [13].

A training session may be initiated on first use of the application or when a new character is first introduced. Once training is finished, a profile of classes and training samples is saved as an XML document. The profile is a hierarchical container of catalogs (groups of related characters, e.g. digits, Latin letters), symbols, writing styles, and training samples [13]. Training samples are divided by different forms for symbols (e.g. a one-stroke versus a two-stroke numeral “ ϕ ”), since many recognition methods are sensitive to the direction and order of strokes. The training extension also provides an interface for the user to manage profiles of training samples.

3.3 The Mathematical Recognition Extension

The mathematical recognition extension is an ongoing project. The objective is to have domain-specific recognizers that handle different kinds of digital ink input. We currently focus on the classification of handwritten mathematical symbols, as the essential component of math formula recognition, and spatial analysis of characters. In our classification paradigm, each character is represented as a

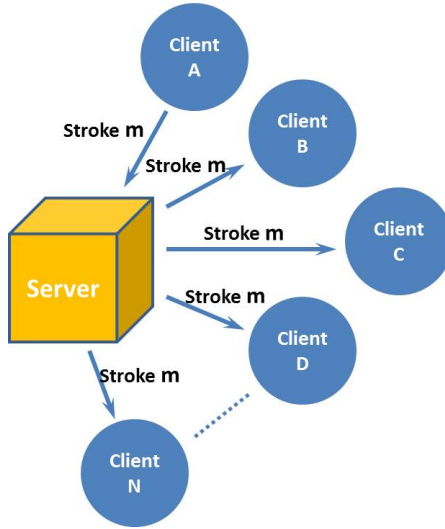


Fig. 6. A client-server configuration with a stroke initiated by a client

single point in a space of curves. The coordinates of this point are the coefficients of truncated orthogonal series approximating the coordinate functions of the trace [14]. Classification of a character is based on the distances to the convex hulls of nearest neighbours in this space. A sequence of incoming strokes is divided to form characters with the highest classification confidence. Spatial analysis of samples is performed based on the relative positioning of the centers of mass of adjacent characters.

This approach typically does not require many training samples to discriminate a class. However, because there is a large number of classes, the training dataset may contain tens of thousands of characters. The dataset evolves with each recognized sample. Synchronization of the evolving per-user training exemplars across several pen-based devices may become tiresome. To address this aspect, the storage of the training database can be delegated to a cloud. [13]. At present, recognition is always done locally, although this could alternatively be done by a server. Locally, the recognition extension accepts raw ink from InkChat and preprocesses it. This preprocessing typically includes computing approximation of the character and normalization with respect to the size and position. Consecutive strokes are merged into candidate symbols and approximated, yielding points in the space of curves. The coefficients are recognized [14] by the recognition extension. A ranked list of recognition results is sent to InkChat for display and confirmation or rejection/selection. The result is stored remotely as classification experience for subsequent adjustment of training clusters.

3.4 The Compression Extension

The compression extension implements a hybrid of functional [15] and linear [16] approximation. There are several schemes for segmentation of digital ink for

compression by the functional approximation method. The most robust is the adaptive segmentation that dynamically selects the degree of approximation and the size of coefficients. Coefficients are recorded as floating-point numbers with base 2 [15]. The functional approximation is best-suited for curly handwriting, for example as with math symbols.

Linear approximation allows compact representation of nearly linear segments, and is therefore suitable for many forms of representation of mathematical and engineering knowledge, such as graphs, tables, or diagrams. The method removes points that least affect the shape of the curve, so long as the error between the original and the approximating curves remains within a threshold. The method can be viewed as a dynamic adjustment of the density of points, depending on the shape of the stroke. The method is very fast and yields reasonable compression.

3.5 The Rendering Extension

Different rendering styles are required for different drawing and writing activities. For example digital painting and Chinese calligraphy may require an instrument that behaves like a paint brush, while diagramming may best be done with an instrument that behaves like a pen or pencil. To support these needs, different brush models may be selected.

Round Brush. The round brush, as its name suggests, draws each ink point as a filled circle. We use three parameters to model the round brush: x , y to indicate the position and r to measure the circle radius which can be a function of the pen tip pressure. We fill the gaps between the circles of consecutive points by forming envelopes with circle tangents, as shown in Figure 7(a).

Tear Drop Brush. The idea of the tear drop brush is to model the contact of a brush head as it varies in distance to the canvas and is dragged along. This contact area is modelled as tear drop – a circle and the area enclosed by the tangents to a trailing point (which may degenerately be on the circumference). Varying the radius models what happens when the brush varies in distance to the canvas, and the trailing point is determined by the past history of the brush. There are five parameters, as shown in Figure 7(b): x and y give the position of the ink point, r gives the head radius, θ the direction of the tail, and ℓ the length of the tail from (x, y) . The tear drop brush model has been adopted by InkML [6] and the calculation of r , θ and ℓ from the stream of x, y and pressure values has been discussed in [17][18]. As with the round brush, strokes are rendered by filling brush shapes and the area formed by tangents between successive brush outlines.

3.6 The Archival Extension

InkChat stores handwritten input using InkML format. Strokes are converted to an InkML stream as they are captured. They are then saved to the current ink session before being sent to other participants. As the stream is received by

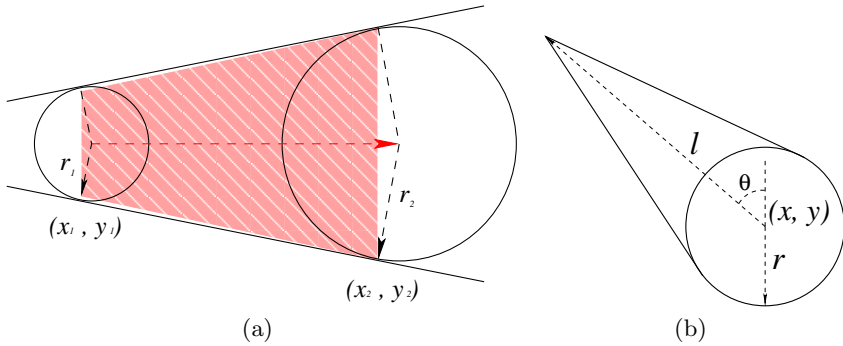


Fig. 7. The brush models: (a) the round brush and (b) the tear drop brush

a participant, InkChat immediately parses the stream and saves the strokes to the current ink session. When the conversation is finished or the user requests to save, InkChat writes the current ink session to an InkML file with the ink from all participants. InkChat also supports loading collaboration sessions from InkML documents.

4 Implementation

We have used the framework presented in section 3 to build the InkChat application. Most components are implemented in Java as it provides strong portability and applications can run on any Java Virtual Machine regardless of system architecture. For platforms that do not support Java directly, the Java code can be compiled to C. Our goal to maintain a single, coherent source that can be compiled for all platforms. Below we briefly describe the implementation of the major components.

4.1 User Interface

The InkChat user interface is illustrated in Figure 8. It is designed to have buttons grouped so that the distance of moving the pen is minimized. Users can write, erase, and highlight by selecting corresponding brushes. Editing is also supported. Users can redo, undo, cut, copy, and paste different kinds of content, such as images, typed-text, and digital ink. In order to better support collaboration, we have provided a floating pointer and a page navigator. The floating pointer can be used to point at target objects on the shared canvas without leaving ink. Together with voice channel, this allows pointing to and discussing aspects of the common canvas. The page navigator allows participants to create new pages or review earlier pages. Once a session is finished, all pages can be recorded and stored for later playback.

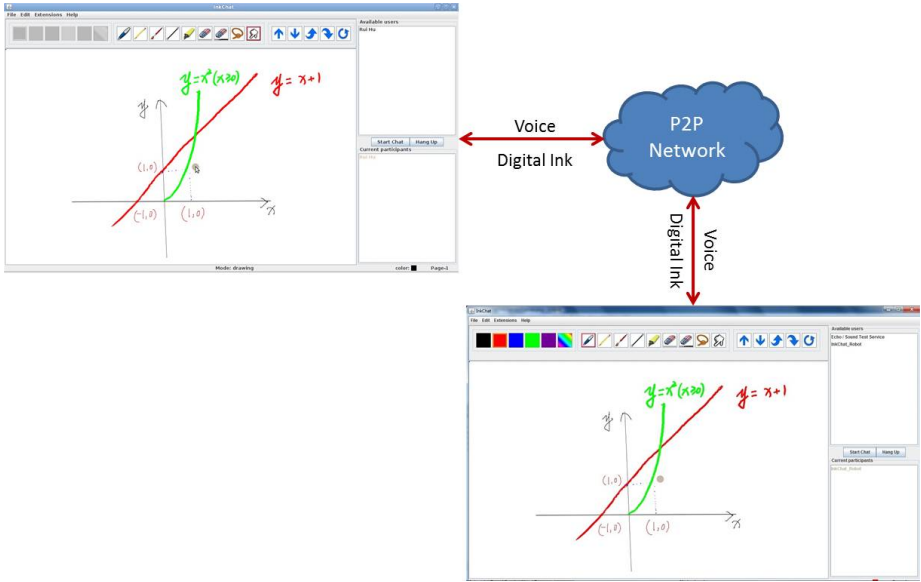


Fig. 8. An example of diagramming in InkChat with Skype service

4.2 Collaboration

The InkChat software combines digital ink and voice in a multi-user conversation with a shared canvas. It is structured so it can use the popular Skype and Google Talk services for transmitting data streams. InkChat can be used by people working together in the same room or on opposite sides of the planet. Sessions may be archived to be resumed later or for later processing. An example of collaboration using InkChat is illustrated in Figure 8. Two participants, the author and user `InkChat_Robot`, are involved in the collaboration using Skype. The author is working on a Windows machine while the `InkChat_Robot` is on a Linux machine. The top left screenshot shows the `InkChat_Robot`'s canvas and the bottom right screenshot shows the author's canvas. Remarkably, the `InkChat_Robot` is using the floating pointer to point around coordinate $(1,1)$. This is streamed to the author's canvas in real time.

4.3 Training

The training extension presents an extensible catalog of symbols and styles, organized in a tabbed panel, as shown in Figure 9(a). Each tab contains a list of symbols. Once the user selects a symbol, the panel with styles becomes available. Styles are shown as animated images for visualization of stroke order and direction. Each sample is associated to an existing or new style. If a style has not been selected, it is determined automatically based on its shape and the number of strokes. All the elements of the interface (catalogs, symbols, styles

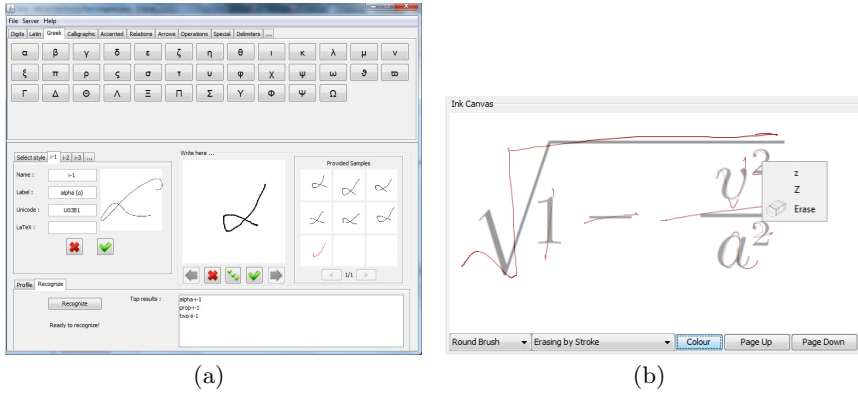


Fig. 9. The recognition interface: (a) the training extension and (b) the mathematics recognition extension

and samples) are dynamic: A context menu is allows the user to create, delete or merge elements. A profile can be saved locally or synchronized with a server.

4.4 Mathematical Recognition

Classification of symbols takes place when a user performs handwritten input through InkChat and the recognition extension is enabled. For each character, a context menu is available that lists the top recognition candidates, see Figure 9(b). If the user chooses another class from among the candidates listed in the context menu, adjacent characters can be reclassified based on the new context information. There are two usual approaches to expression recognition: character-at-a-time (each character is recognized as it is written) and expression-at-a-time (characters are recognized in a sequence, taking advantage of the context) Our current approach lies between these: we are experimenting with recognition within a moving window of context and consider all ink outside that window to be “dry” and recognized. Classification results can be displayed super-imposed on the digital ink or can replace it.

5 Discussion

5.1 Scenarios

Here we describe several cases in which the framework has been found useful.

Online Learning and Tutoring. Figure 10 shows the triangle inequality being discussed by three persons. The top-right panel lists the available friends of the user and the lower-right panel shows the participants who are currently in the session (excluding the user). The Google Talk service is employed in this session. The triangle was drawn by the participant Robot1 InkChat. It was later annotated by the user to better explain the triangle inequality.

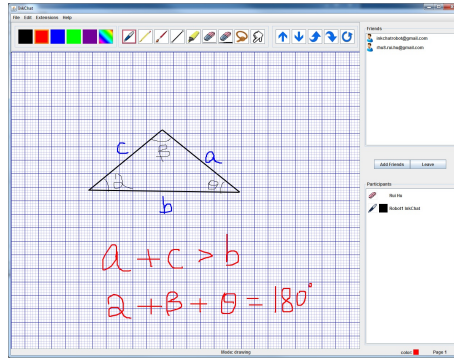


Fig. 10. An example of online learning and tutoring

Collaborative Work on Math and Diagrams. The framework provides a collaborative environment where participants can interact through a shared canvas and a voice channel. Figure 10 shows examples in different domains. The ink and meta-information is shared between participants through Skype or Google Talk.

Demonstration of Animated Diagrams. At least in some cases, animation can significantly improve understanding of certain complex diagrams [19]. The InkChat framework allows digital ink to be animated, reproducing the original writing sequence, or sequenced in some other manner. Thus, animated sketched diagrams can be created and shared in real-time or made available for download.

5.2 Lessons Learnt

The architecture we have presented is a multi-year ongoing project, and there are several important lessons that we have learnt

- **Lesson 1:** *Anticipate that the world will keep changing.* It is important to have the data streams for an application well specified so new technologies, such as JavaScript and HTML 5 canvas can participate.
- **Lesson 2:** *There is power in less capable, but more uniform, interfaces.* We initially had different interfaces for ink sharing, recognition and the other behaviours. Having a common interface, while not perfectly adapted to any of these tasks, allowed them to be combined flexibly in useful and unanticipated ways (e.g., combining the recording/playback and recognition modules).
- **Lesson 3:** *Plan on sharing.* The extensions and the main platform should be designed to allow sharing of the functionality with third parties through the mechanism of web services. For example, the recognition extension should be able to accept a SOAP message with coefficients of a character to be classified and send back the recognition candidates.
- **Lesson 4:** *Don't pin the user down.* Plan on individuals making use of many devices. This implies a protocol for network storage of user-specific information, such as personalized handwriting recognition data.

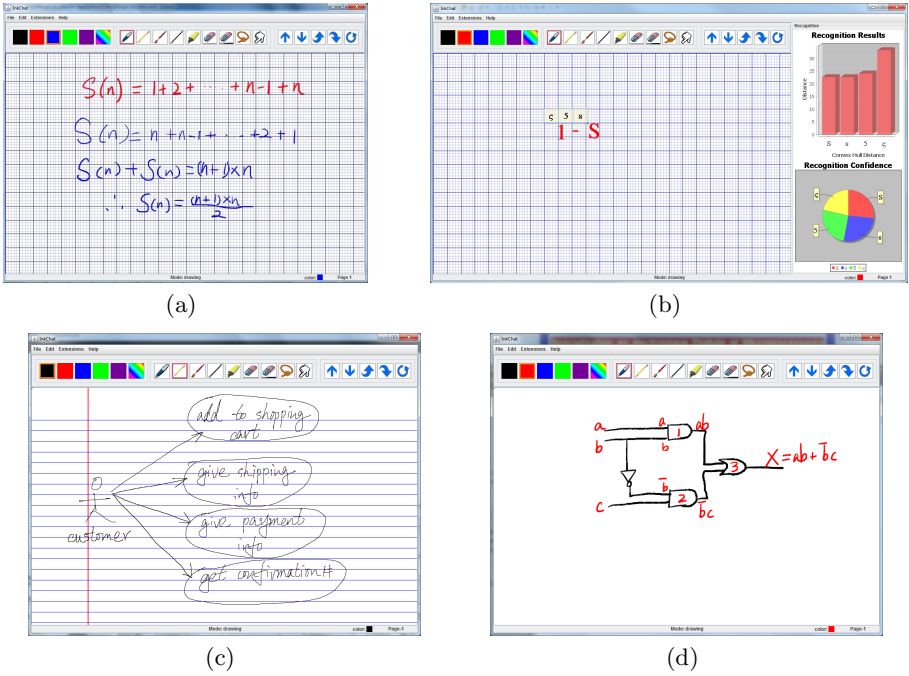


Fig. 11. Examples of math and diagram collaboration (a) formula induction, (b) math symbol recognition, (c) use case diagram, (d) circuit diagram

- **Lesson 5: Anticipate standards.** Tracking draft standards prior to their maturity, and indeed participating in the standardization process, can be an effective strategy to achieve interoperability. The project was using InkML before it was recommended as a standard by W3C. Today InkML allows cut-and-paste of digital ink between applications, e.g. between Microsoft Office 2010 and InkChat.

6 Conclusion and Future Work

We have presented a framework for pen-based, multi-user, online collaboration applicable to mathematical domains. The framework supports teamwork on scientific and engineering problems by allowing participants to make contributions to a shared canvas while having a discussion. The framework is portable and uses InkML, a W3C standard, as a representation format. The architecture of the framework is extension-based with InkChat as the main component. On top of InkChat, several extensions have been developed for collaboration, training, mathematical recognition, compression, rendering and archival. We have presented a high-level overview of these components. In ongoing work, we are particularly interested in improved recognition of mathematical formula and architecting more powerful combinations of extensions.

References

1. Gowers, T., Nielsen, M.: Massively Collaborative Mathematics. *Nature* 461, 879–881 (2009)
2. Maplesoft Inc.: Maple user manual
3. Microsoft Inc.: Onenote (2010)
4. Iotum Inc.: Calliflower, <http://www.calliflower.com/>
5. Dabbleboard Inc.: Dabbleboard, <http://www.dabbleboard.com>
6. Watt, S., Underhill, T.: Ink Markup Language (InkML), <http://www.w3.org/TR/InkML/>
7. Microsoft Inc.: Skype, <http://www.skype.com/>
8. Google Inc.: Google Talk, <http://www.google.com/talk/>
9. Regmi, A., Watt, S.M.: A Collaborative Interface for Multimodal Ink and Audio Documents. In: Proc. 10th International Conference on Document Analysis and Recognition, ICDAR 2009, pp. 901–905. IEEE Computer Society (2009)
10. Ning, H., Williams, J.R., Slocum, A.H., Sanchez, A.: InkBoard - Tablet PC Enabled Design oriented Learning. In: Proc. 7th IASTED International Conference on Computers and Advanced Technology in Education, CATE 2004, August 16-18, pp. 154–160. ACTA Press (2004)
11. Beavers, J., Chou, T., Hinrichs, R., Moffatt, C., Pahud, M., Powers, L., Eaton, J.V.: The Learning Experience Project: Enabling Collaborative Learning with ConferenceXP. Technical Report MSR-TR-2004-42, Microsoft Research (2004)
12. Microsoft Inc.: Ink Serialized Format Specification, [http://download.microsoft.com/download/0/B/E/OBE8BDD7-E5E8-422A-ABFD-4342ED7AD886/InkSerializedFormat\(ISF\)Specification.pdf](http://download.microsoft.com/download/0/B/E/OBE8BDD7-E5E8-422A-ABFD-4342ED7AD886/InkSerializedFormat(ISF)Specification.pdf)
13. Mazalov, V., Watt, S.M.: Writing on Clouds. In: Jeuring, J., et al. (eds.) CICM 2012. LNCS (LNAI), vol. 7362, pp. 402–416. Springer, Heidelberg (2012)
14. Golubitsky, O., Watt, S.M.: Distance-Based Classification of Handwritten Symbols. *International J. on Document Analysis and Recognition* 13(2), 113–146 (2010)
15. Mazalov, V., Watt, S.M.: Digital Ink Compression via Functional Approximation. In: Proc. 12th International Conference on Frontiers in Handwriting Recognition, ICFHR 2010, November 16-18, pp. 688–694. IEEE Computer Society (2010)
16. Mazalov, V., Watt, S.M.: Linear Compression of Digital Ink via Selection of Subsets of Points. In: Proc. 10th IAPR International Workshop on Document Analysis Systems, DAS 2012, March 27-29, pp. 429–434. IEEE Computer Society (2012)
17. Hu, R.: Portable Implementation of Digital Ink: Collaboration and Calligraphy. Master’s thesis, The University of Western Ontario, Canada (2009)
18. Watt, S.M.: On the Mathematics of Calligraphy (invited talk). In: International Conference on Mathematics Mechanization – In Honor of Professor Wen-Tsun Wu’s Ninetieth Birthday, Beijing, China (2009)
19. Burd, E., Overy, D., Wheatman, A.: Evaluating Using Animation to Improve Understanding of Sequence Diagrams. In: Proc. 10th International Workshop on Program Comprehension, IWPC 2002, p. 107. IEEE Computer Society (2002)

Cost-Effective Integration of MKM Semantic Services into Editing Environments

Constantin Jucovschi

Jacobs University Bremen, Germany

Abstract. Integration of MKM services into editors has been of big interest in both formal as well as informal areas of MKM. Until now, most of the efforts to integrate MKM services into editing environments are done on an individual basis which results in high creation and maintenance costs.

In this paper, I propose an architecture which allows editing environments and MKM services to be integrated in a more efficient way. This is accomplished by integrating editors and services only once with a real-time document synchronization and service broker. Doing so simplifies the development of services, as well as editor integrations. Integrating new services into an arbitrary number of already integrated editors can then take as little as 3-4 hours of work.

1 Introduction

Integration of MKM services into editors has been of big interest in both formal as well as informal areas of MKM. In the formal area of MKM, it becomes more and more important to have an efficient way to work with formal documents and pass information between interactive provers and the user. Examples of useful services are: showing type information when hovering over an expression, navigating to definitions of symbols, and supporting editor features like folding inside long formulae.

Informal MKM needs editing support to make it easier for authors to create semantically annotated documents. This can mean integration of Natural Language Processing (NLP) services to e.g. spot mathematical terms as well as hiding (or folding) existing semantic annotations in order to provide a better reading experience.

Until now, most of the efforts to integrate MKM services into editing environments were done on an individual basis, namely, some service X was integrated into an editing environment Y . Obviously it would have been more efficient to integrate service X into all editing environments Y where this service makes sense. The problem is that developing and maintaining such integrations requires a lot of effort and hence MKM services are usually integrated with maybe one or two editing environments.

The high cost of semantic service integration has an especially negative impact on informal MKM because its users, the authors of mathematical texts, write mathematics in a handful of different programs and operating systems. Even if

one chooses to support the 5 most used editing environments, it would still be too expensive to maintain 10 semantic service integrations. Clearly, a different integration strategy is called for.

In this paper, I propose an architecture allowing tight integration of authoring services into the authoring process. It enables distributed services, running on different platforms and hardware, to be notified of changes made by the user to a certain document. Services can then react to these changes by modifying/coloring document's text as well as enriching parts of the text with extra (invisible to the user) semantic annotations. This architecture can accommodate both 1. reactive services like syntax highlighting giving user the illusion that the service runs natively in the editor as well as, 2. time-consuming services like Natural Language Processing tasks, without requiring the user to wait while text is processed.

In the next section I discuss in more detail what type of services would fit the presented framework, and compare it to other existing MKM frameworks. In section 3, I present the proposed integration architecture. To validate the architecture I implemented four services and integrated them in two editors. In section 4, I briefly describe these services, and give some high level technical details on how new services can be built. Section 5 concludes the paper.

2 Aims and Scope of Integration

The task of tool integration is a very complex and multi-faceted one. Many frameworks and technologies [Wic04] have been proposed to integrate tools, each optimizing some aspects of tool integration and yet, none of them is widely adopted. The current paper does not attempt to create a framework to integrate all possible editors with all possible services. It considers only pure text editors and integrates only services that participate in the editing/authoring process. The scope of the framework is purposefully kept relatively small so that it solves a well-defined part of editor-service integration problem and can eventually be used along with other integration frameworks.

In the next section, I would like to make more explicit the types of services that are included in the scope of the framework. In section 2.2 I analyze what integration strategies will be used to achieve optimal integration. This will later help me in section 2.3 to differentiate the framework more clearly from other existing frameworks.

2.1 Targeted Authoring Services

In this paper by “authoring service for text-based documents” (abbreviated as “authoring service” or just “service”), I mean a service which provides some added value to the process of authoring the text document by:

1. reacting to document changes and giving feedback e.g. by coloring/underlining parts of text,

2. performing changes to the document e.g. as result of an explicit request to autocomplete, use a template, or fold some part of text/formula.

The strategy to achieve editor-service integration is to require authoring services to be agnostic of advanced editor features like the ability to fold lines or embed images/MathML formulae. Services should only be allowed to assume a relatively simple document model that allows a limited set of operations (both described in section 3.2) and which are supported by most editing environments. Likewise, services should only assume a limited set of possible interactions with the user (see section 3.4).

A way to decide whether a certain service is in the scope of the presented integration framework is to analyze whether it can be realized conforming to the limited document model and interaction possibilities.

2.2 Levels of Integration

One of the classical ways used to describe and compare integrations was proposed by Wasserman [Was90] and later improved by Thomas and Nejmeh [TN92]. They proposes 4 dimensions along which an integration can be analyzed, namely: presentation, data, control and process integration dimensions. Presentation integration accounts for the level at which tools share a common “look and feel”, mental models of interaction, and interaction paradigms. Data integration dimension analyzes how data is produced, shared and kept consistent among tools. Control integration dimension analyzes the level to which tools use each other’s services. The process integration dimension describes how well tools are aware of constraints, events and workflows taking place in the system.

Note that high or low level of integration does not reflect the quality of some integration. Low integration level in some dimension only suggests that those tools can be easily decoupled and interchanged. High integration level, on the other hand, suggests that tools connect in a deeper way and can enable features not possible otherwise. Experience suggests that, low level of integration require less maintenance costs in the long run and should be used whenever possible.

According to description of the targeted authoring services, I derived the integration levels that need to be supported in each integration dimension. Namely we need:

1. medium-high presentation integration level, because services need to be able to change the text/colors of the document as well as interact with the user using some high level interaction paradigms. While the type of changes/interactions a service can perform are limited, a service should be able to perform these actions unrestricted by other components.
2. high data integration level due to the fact all service are distributed and still need to be able to perform changes to the edited document. Hence synchronization and data consistency mechanisms are needed.
3. low control integration as the framework should only give support for integrating services with editors. Service-service integrations are outside the scope of integration.

4. low process integration mainly due to the fact that services are expected to be mostly stand-alone and the workflows should only involve an editor and a service.

2.3 Comparison to Other MKM Integrations

A lot of integrations combining several tools have been developed in MKM. In the context of the current paper only several types of integrations are interesting:

1. frameworks that enable integration of services into authoring process in a consistent, service independent way,
2. integrations which allow multiple services to listen/react to document changes.

The Proof General (PG) framework along with the PGIP message protocol [ALW07] constitute the base for several popular integrations between editors (e.g. emacs, Eclipse) and interactive provers like Isabelle, Coq and HOL. PG framework differs from presented framework in the following ways:

1. low presentation integration — consisting of changing the color of text regions according to prover state, accompanied by locks of regions under processing. The later is not a typical interaction a user would expect. Integrated provers cannot directly influence the display or interact with the users. Hence only the broker component is directly integrated from a presentation integration point of view.
2. low data integration — editors and services communicate mainly by sending parts of the source text to the prover as result of the user changing regions of text. This data never gets changed by services and hence no synchronization or consistency checking is needed.
3. medium control integration — the broker uses different protocols to talk with displays and provers. Additionally, it has the role of orchestrating interaction between them.

The effort of Aspinall et al. [ALW06] further extends PG architecture to integrate rendering processors (e.g. \LaTeX) and possibly other tools (e.g. code generators). These newly integrated components are loosely integrated by running them on a (hidden from the user) updated version of the original document. The broker-prover integration becomes tighter due to the documentation and script backflow mechanisms. Differences to the current framework are:

1. from the presentation integration perspective it extends the PG framework by two (or more) additional views of the document. These views are presented and updated in separate windows and provide the user with more focused views on the authored content. These new views do not seem to provide additional interactions to the user. In conclusion, the presentation integration is tighter compared to PG but still relatively low.
2. data integration becomes tighter between the broker and prover components as a result of the backflow mechanisms but is still relatively low. The additional complexity due to the backflow is mostly in the broker component

which needs to know where in the central document to integrate data coming from the prover. Data passed around still never gets modified and hence no advanced synchronization or consistency checking is needed.

3. control integration is similar to the PG framework.

The integration of provers and editors is done quite differently in PIDE [Wen10]. Instead of fixing a protocol encapsulating all the features a prover can provide (like PGIP does), a document model is specified and the protocol to interact with that document model is fixed. This has the advantage that editors need to know much less about the provers features, and only need to provide them with changes the user made to the document. Conceptually this is very similar to the approach taken in the current work. The difference is that in the case of PIDE, the document model is shared only between two entities (the editor and the prover) and that these entities share the same running environment. The current architecture allows several authoring services to listen and change the shared document and allows them to be distributed.

3 Editor Service Integration Architecture

Creating and maintaining integrations between software programs is generally an expensive task. However, there are some well known best practices for these tasks which have been proven to help a lot in reducing both creation as well as maintenance costs. A good example is the integration between database systems and hundreds of languages and frameworks. Some of the key aspects that make such integrations possible are:

- P1. Client-server architecture** allows clients and servers to be developed and executed on arbitrarily different environments.
- P2. Stable communication protocol** on the server side reduces maintenance costs and makes documentation more stable and complete.
- P3. Declarative API** is usually more stable as it requires definition of a small set of primitives and some way to compose them.

The goal of the current architecture is to integrate m services into n editors. Direct integration between editors and services (Figure 1a) would require $n \cdot m$ integrations to be implemented. To reduce this number, I propose to create an independent Real-Time Document Synchronization and Service Broker (ReDSyS) component, as shown in Figure 1, which complies with the practices **P1-P3**, and which integrates with each service and editor exactly once in a client-server manner (ReDSyS being the server). In this way we only need $n+m$ integrations. The ReDSyS server API is expected to be stable (requirement P2) hence any upgrades of editors/services may require adjustments in the integration only on the editor/service part.

Section 3.1 describes in more detail the ReDSyS component and how it integrates with editors and services. Section 3.2 presents the shared document model and shows how editing changes are represented. In section 3.3, I discuss management of change issues that can appear when integrating time consuming

services as well as make explicit some requirements for reactive services. Section 3.4 presents the interaction model between users, editors and services.

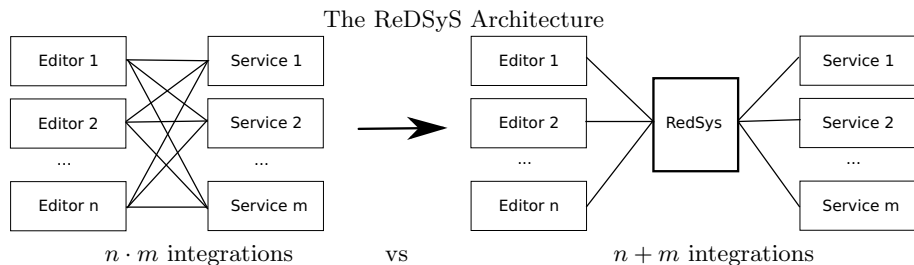


Fig. 1. Direct integrations between editors and environments vs indirect integration through ReDSyS component

3.1 The Real-Time Document Synchronization and Service Broker

The ReDSyS component has two main responsibilities:

1. provide a way for editors to trigger events to all or some subset of services,
2. allow editors and services to independently add/remove meta-data or text to the shared document in real-time.

The first responsibility is typical for broker components. In our case, it makes it possible for editors to request autocompletion suggestions (from all services) or request the type of a symbol (from e.g. Twelf services for LF documents [Pfe91]).

The second responsibility enables services to run independently, distributed on different systems and parallel to the editing process. So while the user is typing, a service might decide to start processing and integrate the results back into the document when finished. Whenever a user changes the document, services get notified by the ReDSyS component and each service has the freedom to decide whether to interrupt current processing (if available) or not. Thanks to real-time document editing solutions, it is quite often possible to automatically merge service results computed on older versions of the document into the current version.

To understand the interaction between editors, services and ReDSyS better, let me describe how \LaTeX [Koh04] editing in Eclipse [Ecl] can be integrated with a term spotting and an autocompletion service. You can follow the communication between components in Figure 2.

The first step is to install the “ \LaTeX -padconnector” plugin into Eclipse which, upon opening an \LaTeX file, uploads it to the ReDSyS component and opens it in a typical Eclipse editing window. The opened document is, in fact, the shared document. The ReDSyS architecture takes care of starting the semantic services.

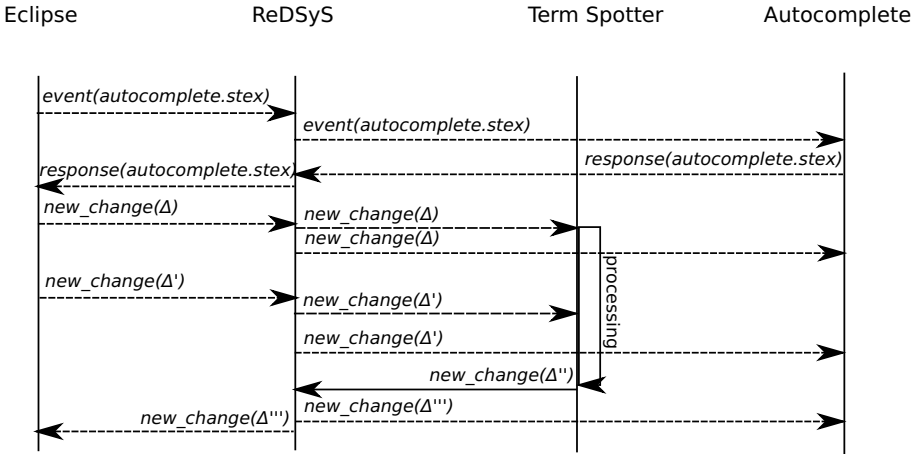


Fig. 2. Communication diagram among architecture components after an autocomplete requests followed by term spotting

The “ \LaTeX -padconnector” plugin is programmed so that, when the user presses ctrl+space, it synchronously notifies (i.e. waits for the result) the ReDSyS architecture that an event with a predefined URI “autocomplete.stex” took place and passes the current cursor coordinates as parameters. The autocomplete service catches the event and based on the parameters decides on autocomplete suggestions. The editor receives autocomplete suggestions from ReDSyS and displays them.

Whenever the user changes the document, a changeset (or diff) is computed and sent to ReDSyS. This passes on the changeset to all the other services so that they have an up-to-date version of the document. Let us suppose that the Term Spotter service decides to start a relatively complex NLP processing task for of the new version (e.g. version 40) of the document. While processing, some other changeset comes to the Term Spotter service but it decides not to cancel the NLP task. The shared document has now version 41 but the NLP task has computed a changeset (Δ'') which highlights new found terms based on document version 40. The Term Spotter service sends change Δ'' to ReDSyS also including information about the document version on which the changeset is based on (i.e. 40). The ReDSyS component tries to merge the changes and if it succeeds, sends a merged changeset to all the other components.

3.2 Document Model and Changesets

In this section I introduce the shared document model as it is important to understand what kind of information is shared among services and how. This document model is the same as that of the Etherpad-lite system [Eth].

We define a finite alphabet $A = \{\alpha_1, \alpha_2, \dots, \alpha_m\}$. A string $s_A = c_1c_2\dots c_n$ is a finite sequence of characters from alphabet A and length $len(s_A) = n$ is defined

as the number of characters in that sequence. Let Str_A be the set of strings over alphabet A . For the sake of simplicity we will omit alphabet A in notations when it can be unambiguously inferred from the context.

An attribute pool $AP = \{(id, (key, val)) \in \mathbb{N} \times (Str \times Str) \mid id - \text{is unique}\}$ is a set of key-value pairs which also have a unique id assigned to each of them. This defines a function $attmap : \mathbb{N} \rightarrow (Str \times Str)$ returning the key-value pair associated to a certain id.

A document $d = (text, attmap, att)$ where $text \in Str$ represents the text in that document, $attmap \in \mathbb{N} \rightarrow (Str \times Str)$ is the function associated to an attribute pool and $att : \mathbb{N} \rightarrow Set(\mathbb{N})$ with $att(i) = S, \|S\| < \infty$ specifies which set of attributes are assigned to character $i, 0 \leq i < len(t)$ in document's text.

A change operation $d = (op, S, len, t) \in \{+, -, =\} \times Set(\mathbb{N}) \times \mathbb{N} \times Str$ either

1. inserts (op="+") text t of length len and applies attributes in S to each of the inserted characters, or
2. deletes (op="-") len characters (and their attributes) from a text ($t = ""$ and $S = \emptyset$), or
3. leaves unchanged (op="=") len characters but applies attributes in S (if $S \neq \emptyset$) to each of them.

Editing changes inside a document are represented using lists of change operations $o = \{op_1 op_2 \dots op_k\}$ where op_j are change operations such that no two consecutive operations have the same type and attribute sets (otherwise we can join them together).

A changeset is defined as $c = (l, l', attmap, o)$, where l is the length of the text before the change, l' is the length of the text after the change, $attmap$ contains the updated attribute pool (only new elements allowed) and a sequence of change operations to be applied on the text.

To provide a better intuition for these notions, consider the document

$$d = \left(\text{"Math is great"}, \left\{ \begin{array}{l} 0 \rightarrow (\text{"bold"}, \text{"true"}) \\ 1 \rightarrow (\text{"author"}, \text{"p1"}) \\ 2 \rightarrow (\text{"author"}, \text{"p2"}) \end{array} \right\}, \left\{ \begin{array}{l} 0 \leq i \leq 4, \{1\} \\ 5 \leq i < 7, \{0, 2\} \\ 7 \leq i \leq 12, \{2\} \end{array} \right\} \right) \quad (1)$$

which says that word "Math" was authored by "p1", the rest of the text by "p2" and the word "is" is bold. Now consider a changeset

$$c = \left(13, 12, \left\{ \begin{array}{l} 0 \rightarrow (\text{"bold"}, \text{"true"}) \\ 1 \rightarrow (\text{"author"}, \text{"p1"}) \\ 2 \rightarrow (\text{"author"}, \text{"p2"}) \\ 3 \rightarrow (\text{"author"}, \text{" "}) \end{array} \right\}, \left\{ \begin{array}{l} (\text{"="}, \{3\}, 1, \text{" "}) \\ (\text{"-"}, \emptyset, 3, \text{" "}) \\ (\text{"+"}, \emptyset, 2, \text{"KM"}), \\ (\text{"="}, \emptyset, 9, \text{" "}) \end{array} \right\} \right) \quad (2)$$

which when applied to d , would change the text to "MKM is great", the word "MKM" would have no author and the rest remains unchanged.

3.3 Time Consuming vs. Reactive Services

The ReDSyS architecture can support both time consuming services (e.g. NLP tasks) that should not hinder the user from further editing of the document, as well as services that need to give user the impression that the service is running natively inside the editor (e.g. syntax highlighting).

As described in the communication workflow in the previous section, time consuming services can start processing at any given point in time (e.g. version 40) and integrate their results automatically (e.g. at version 50) by creating and sending a changeset (call it X) based on the version when processing started (i.e. 40). The ReDSyS component has some merging strategies to integrate such changes but they will fail if the changes done in between i.e. from version 41-50 overlap with areas in changeset X.

This default behavior can be improved in a number of ways. First, a time consuming service is still notified of changes done to the document even while it is processing. Hence, it can interrupt the processing or restart it if the document was changed in the area currently under processing. In this way, computing power to finish the processing (and then realize that it cannot be integrated) is not lost. The second solution is to try to incorporate incoming changes while or at the end of processing and ultimately create a changeset based on the latest version of the document. Hence a time consuming service is responsible for its own management of change.

To give users the feeling that reactive services run natively in the editor, they need to be very optimized both in speed and in the size of the changesets they produce. These services might need to run at a rate of 20 times a second in order to accommodate several users editing in the same time. Hence it is very important that reactive services can cache results and start processing a document without reading all of it. Also the changesets that reactive services produce should be small and only change areas of the document that really need changing. For example, a bad syntax highlighting service that creates a changeset recoloring the whole document (and not only the parts the need recoloring) could invalidate the processing of all the time consuming services.

3.4 User Interaction Model

In the current framework, services and editors no longer integrate directly but they still need to interact, e.g. services might need to ask the user to disambiguate a mathematical term. Such interactions must be standardized so that all editors ask services to perform a certain action (e.g. autocomplete) in the same way.

Every type of interaction between users, editors and services has a predefined URI. An example of such URIs is “autocomplete.stex” which, when broadcasted by the ReDSyS component to the services, expects them to return \LaTeX based autocompletion suggestions and then displayed to the user. Another example is “contextmenu.spotter_plugin.10”. This URI can be used inside an attribute (“ui”, “contextmenu.spotter_plugin.10”) which, just like the (“bold”, “true”) attribute, can be applied to some part of the text in the shared document model presented in section [3.2](#).

When the editor sees attributes having key “ui” and value prefixed with “contextmenu.”, it knows to display a context menu when the text having this attribute is right-clicked. The menu items in the context menu are fetched from the “spotter_plugin” component (i.e. the Term Spotter) and “10” is passed as an argument to identify which context menu should be displayed.

It is the editor that is responsible to understand interaction URIs and act accordingly. That is why it is important to define interactions in a general and reusable manner so that many services can take advantage of it. Currently, the set of predefined interaction URIs is relatively small and fits the use cases presented in the implementation section. However, it certainly needs to be revised and extended to fit a more general range of interactions.

4 Architecture Implementation

To validate the presented architecture and to prove its applicability to a wide range of semantic services and editors, I chose to integrate four \LaTeX semantic services into two editors. The semantic services were picked to address different integration issues and are described in more detail in Section 4.1. Section 4.2 gives a glimpse into the APIs that need to be used in order to create a service. Finally, section 4.3 discusses extensibility and reuse issues of the architecture.

The editors I used to validate the architecture are: Eclipse (desktop based editor) and Etherpad’s Web Client (web-based editor). To support the Eclipse based editor I implemented my own synchronization library with the ReDSyS component called jeasysync2. The real-time document sharing platform (ReDSyS) used in my implementation is an extension of the Etherpad-lite system and can be found at <https://github.com/jucovschi/etherpad-lite/tree/mkm>.

4.1 Implemented Semantic Services

My architecture can be seen as an enabler for user-editor-service type of interactions and hence this is the part which needs most testing. Let us consider the service of semantic syntax highlighting. The user-editor-service interaction consists in the service being able to change the color of text. Testing a more complex service, requiring coloring parts of text does not make sense because the additional service complexity is independent of the presented architecture. Hence I chose services testing different aspects of user-editor-service interaction, namely:

\LaTeX Semantic Syntax Highlighter colors \LaTeX code based on its semantic meaning. This is a service which cannot be implemented using regular expressions — the main tool for syntax highlighting in many editors. Hence I implemented it as a service and integrated it in editors via ReDSyS. Even though it only needs to highlight text, it has to do that more often than most other services hence it helps benchmarking the user-editor-service interaction speed.

Term Spotter is a NLP based service which tries to spot mathematical terms inside a document. The interaction with the user is very similar to that of spell checking, namely, spotted mathematical terms are underlined while the user is typing. The user can then choose to add semantic references to spotted terms. This is an example of a service with heavier server side part and helps us test how service results are automatically integrated (if possible) into newer document versions.

TermRef Hider and Transclusion services are examples of advanced editing features, one hiding parts of annotations from the user and other showing referenced text instead of references. As both services showcase very important results of using proposed architecture I address them in more detail in the next section.

Support for Advanced Editing Features

Inline annotated documents are very hard to author because they contain additional implicit knowledge that 1. is redundant to the author as she already knows it and 2. hinders a clear reading experience. Stand-off or parallel annotations solve these problems but they require the use of special editing environments every time a small update needs to be performed. Failing to do so may invalidate existing annotations. Generally MKM systems only support inline annotations on documents that are editable by users.

In Figures 3 and 4 compare a mathematical document to its semantically annotated version. The difference between the readability of these documents is quite obvious even though all we did was to annotate three terms (using `\termref` macros) and do four transclusions (using `\STRlabel` and `\STRcopy` macros).

To make the text in figure 4 look as readable as the one in 3, we need to support 2 features, namely: inline folding and transclusion. Using inline folding, one could collapse a whole `\termref` to show only the text in its second argument. The transclusion feature would then replace `\STRcopy` references with the text in the `\STRlabel`.

The inline folding or transclusion features are not supported by most editing environments used to author MKM formats like \LaTeX , Mizar [UB06] or LF [Pfe91]. Adding these features directly in each of the authoring environments requires a lot of initial development effort and incurs high maintenance costs when the editor evolves.

4.2 Libraries and APIs

Currently one can create new services for my architecture using either JavaScript or Java programming languages. JavaScript services are implemented in Etherpad-lite's native plugin system. Java services should use the `jeasysync2` library. In both cases services must implement the following interface:

```

1 void init(Changeset initialText, AttributePool pool);
2 void update(Changeset lastChangeset, AttributePool newPool, ChangesetAcceptor csAcceptor);

```

```

1The gravitational potential energy of a system of masses $m_1$ and $m_2$
  at a distance $r$ using gravitational constant $G$ is
3\begin{equation}
  U = -G\frac{m_1m_2}{r}+K
5\end{equation}
  where $K$ is the constant of integration. Choosing the convention that $K=0$
7makes calculations simpler, albeit at the cost of making $U$ negative.

```

Fig. 3. Conventional mathematical document

```

1The \termref{cd=physics-energy, name=grav-potential}{gravitational potential energy}
  of a system of masses \STRlabel[m1]{$m_1$} \STRcopy{m1} and \STRlabel[m2]{$m_2$}
3\STRcopy{m2} at a distance \STRlabel[r]{$r$} \STRcopy{r} using
  \termref{cd=physics-constants, name=grav-constant}{gravitational constant}
5\STRlabel[G]{$G$} \STRcopy{G} is \STRlabel[U]{$U$} \STRcopy{U}
  \begin{equation}
7  \STRcopy{U} = -\STRcopy{G}\frac{\STRcopy{m1}\STRcopy{m2}}{\STRcopy{r}}+
  \STRcopy{K}
9\end{equation}
  where \STRcopy{K} is the \termref{cd=physics-constants, name=integration}{constant
11of integration}. Choosing the convention that \STRcopy{K}$=0$ makes calculations
  simpler, albeit at the cost of making \STRcopy{U} negative.

```

Fig. 4. Semantically annotated mathematical document

The `init` method is called when initializing the service. The first parameter is the `changeset` which, if applied to an empty text, generates the current document (note that attributes are included as well).

The `update` method notifies the service of new updates. This is where the service should decide whether to start/restart processing. The `update` function is called asynchronously in separate threads so special care should be taken not to run into race conditions. The `ChangesetAcceptor` callback allows the service to send `changesets` back to the `ReDSyS` component when processing is finished.

Creating `changesets` is easily done through a utility class called `ChangesetBuilder` with the methods:

```

  void keep(int noChars, AttributeList attribs);
2 void insert(String text, AttributeList attribs);
  void remove(int noChars);

```

This class allows services to specify changes they want to perform in a sequential way e.g. keep the first 10 characters untouched, remove the next 5, insert text “Hello World” and apply attribute [“bold”, “true”] to it, keep the next 2 characters unchanged but apply attribute [“bold”, “”] to them etc. The `ChangesetBuilder` class will produce a correctly encoded `changeset` which can be then transmitted to `ReDSyS`.

The last best practice I would like to share is a simple and efficient algorithm of converting a list of changes of type “*apply attribute [key_i, value_i] from character begin_i to character end_i*” to a sequential list of changes suitable for the `ChangesetBuilder`. I used this algorithm (with minor changes) for all 4 services, hence might be of interest for future service developers.

- we create a list of “event” triples having signature $(type, i, attr)$ where $type$ is either “add” - to add attribute “attr” to the list of attributes applied to all following characters, or “remove” to remove attr from the list of attributes. Index i specifies at which position in the sequence a certain event should take place.
- for each rule of type “*apply attribute [key_i, value_i] from character begin_i to character end_i*” add event triples (“add”, $begin_i$, $[key_i, value_i]$) and (“remove”, $end_i + 1$, $[key_i, value_i]$)
- sort the event list by the i values
- initialize an empty list of attributes called *currentAttrs* and set $lastPos = 0$
- iterate through the sorted event list and let $(type, i, attr)$ be the current event
 - if $i > lastPos$, add sequential operation $keep(i - lastPos, currentAttrs)$ and set $lastPos = i$.
 - if $type = \text{“add”}$, add attr to *currentAttrs*
 - if $type = \text{“remove”}$, remove attr from *currentAttrs*

This algorithm can be generalized to handle events which delete or insert text as well.

4.3 Evaluation of Integration Costs

The aim of the presented architecture is to minimize integration costs and hence in this section I want to evaluate what we gain by using it.

Costs for Integrating Custom Editors / Services

Both editors and services need to implement the following functionality

RE1. Connect to the ReDSyS component,

RE2. Implement document model and changeset synchronization mechanisms.

Both RE1 and RE2 can be reused from other already integrated editors/services. If no such implementations exist for a certain programming language, my own experience shows that one needs to invest about one day for implementing RE1 and about two weeks for RE2. Implementing RE2 requires mostly code porting skills (for about 2k lines of code) i.e. good understanding of particularities of programming languages but does not require deep understanding of the algorithms themselves. Additionally, unit tests help a lot finding and fixing bugs. Currently, RE1 and RE2 are available for Java and JavaScript languages using *jeasysync2*¹ and *easysync2*² libraries.

Editors need to additionally implement styling and user interaction mechanisms based on attributes in the shared document model. Depending on the editing environment, this might take several more days.

¹ <https://github.com/jucovschi/jeasysync2>

² <https://github.com/Pita/etherpad-lite/static/js/Changeset.js>

Integration of services requires implementation of the interaction of the service with the shared document and with the user. To integrate each service described in section 4, I needed 50-100 lines of code.

Requirements for Integrating a Custom Interaction

To add new user interactions, one has to choose a unique URI to identify the interaction and then use this URI from an editor (to trigger events) or from services by using it inside an attribute. The biggest cost associated with adding/extending the set of interactions is that of propagating it to all already integrated editors. Versioning and change management of interaction URIs is an open issue.

5 Conclusion and Future Work

The extent to which editing environments could support the authoring process of MKM documents is far from being reached. There are lots useful authoring services which, if integrated in editing environments, would make authoring of MKM documents easier to learn, more efficient and less error-prone.

Many MKM editing services are available only in certain editing environments but even more services live solely in the wish-lists of MKM authors. An important reason for it is that creating and maintaining such integrations is very expensive. This paper suggests that integration of editors with MKM authoring services can be done in an efficient way. Namely, a service showing the type of LF symbols needs to be integrated with the ReDSyS component once and then be used in a ever-growing list of editors like Eclipse, jEdit or even web-editors. Conversely, once an editor (e.g. \TeX macs) integrates with the ReDSyS component, it would be able to provide the user with all the services already integrated with the ReDSyS component.

The implementation part of this paper allowed me to test my ideas and I found out that integrating a service into all already integrated editors can take as little as 3-4 hours of work. In case that no communication and synchronization libraries are available for a certain language, one can implement them in about 2-3 weeks,

While the focus of the current paper is to reduce the costs for integrating m services into n editors, the suggested solution also:

1. makes is possible for services which need longer processing times to run in the background without interrupting the user's authoring experience,
2. allows services to be implemented in any convenient programming language or framework and even be distributed on different hardware, and
3. extends the typical plain-text document model with attributes which provide a very convenient storage for layers of semantic information inferred by services.

Future research directions include development and integration of new MKM services into editors, extending the list of programming languages which can connect to the ReDSyS component, and integrating additional editors into the proposed architecture.

References

- [ALW06] Aspinall, D., Lüth, C., Wolff, B.: Assisted Proof Document Authoring. In: Kohlhase, M. (ed.) MKM 2005. LNCS (LNAI), vol. 3863, pp. 65–80. Springer, Heidelberg (2006), <http://www.springerlink.com/index/fq4068582k604115.pdf>, doi:10.1007/11618027_5
- [ALW07] Aspinall, D., Lüth, C., Winterstein, D.: A Framework for Interactive Proof. In: Kauers, M., Kerber, M., Miner, R., Windsteiger, W. (eds.) MKM/Calculus 2007. LNCS (LNAI), vol. 4573, pp. 161–175. Springer, Heidelberg (2007)
- [Ecl] Eclipse: An open development platform (December 2011), <http://www.eclipse.org/>
- [Eth] Etherpad lite: real time collaborative editor (February 2012), web page at <https://github.com/Pita/etherpad-lite/raw/master/doc/easysync/easysync-notes.pdf>
- [Koh04] Kohlhase, M.: Semantic Markup for TEX/LATEX. In: Libbrecht, P. (ed.) Mathematical User Interfaces (2004), <http://www.activemath.org/~paul/MathUI04>
- [Pfe91] Pfenning, F.: Logic Programming in the LF Logical Framework. In: Huet, G.P., Plotkin, G.D. (eds.) Logical Frameworks. Cambridge University Press (1991)
- [TN92] Thomas, I., Nejme, B.A.: Definitions of tool integration for environments. IEEE Software 9(2), 29–35 (1992) ISSN: 07407459, <http://doi.ieeecomputersociety.org/10.1109/52.120599>, doi:10.1109/52.120599
- [UB06] Urban, J., Bancerek, G.: Presenting and Explaining Mizar. In: Autexier, S., Benzmüller, C. (eds.) Proceedings of the International Workshop User Interfaces for Theorem Provers (UITP 2006), Seattle, USA, pp. 97–108 (2006)
- [Was90] Wasserman, A.L.: Tool integration in software engineering environments. Development 1(6), 137–149 (1990) ISSN: 02686961, <http://www.springerlink.com/content/p582q2n825k87n15>, doi:10.1007/3-540-53452-0_38
- [Wen10] Wenzel, M.: Asynchronous Proof Processing with Isabelle/ Scala and Isabelle/jEdit. In: Sacerdoti Coen, C., Aspinall, D. (eds.) FLOC 2010 Satellite Workshop User Interfaces for Theorem Provers (UITP 2010). ENTCS, Elsevier (2010), <http://www.lri.fr/~wenzel/papers/async-isabelle-scala.pdf>
- [Wic04] Wicks, M.: Tool Integration in Software Engineering: The State of the Art in 2004. Integration The VLSI Journal, 1–26 (August 2004), <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.59.1969>

Understanding the Learners' Actions when Using Mathematics Learning Tools

Paul Libbrecht¹, Sandra Rebholz², Daniel Herding³,
Wolfgang Müller², and Felix Tscheulin²

¹ Institute for Mathematics and Informatics, Karlsruhe University of Education

² Media Education and Visualization Group, Weingarten University of Education

³ Computer-Supported Learning Research Group, RWTH Aachen University
Germany

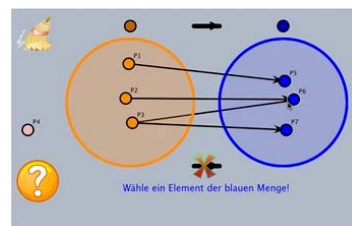
Abstract. The use of computer-based mathematics tools is widespread in learning. Depending on the way that these tools assess the learner's solution paths, one can distinguish between automatic assessment tools and semi-automatic assessment tools. Automatic assessment tools directly provide all feedback necessary to the learners, while semi-automatic assessment tools involve the teachers as part the assessment process. They are provided with as much information as possible on the learners' interactions with the tool.

How can the teachers know how the learning tools were used and which intermediate steps led to a solution? How can the teachers respond to a learner's question that arises while using a computer tool? Little is available to answer this beyond interacting directly with the computer and performing a few manipulations to understand the tools' state.

This paper presents SMALA, a web-based logging architecture that addresses these problems by recording, analyzing and representing user actions. While respecting the learner's privacy, the SMALA architecture supports the teachers by offering fine-grained representations of the learners' activities as well as overviews of the progress of a classroom.

1 Learners' Actions and the Perception of Teachers

Interactive learning tools offer rich possibilities to students when learning mathematics. On the one hand, they offer unprecedented possibilities to explore dynamic representations of the mathematical concepts: they allow students to discover the domain's objects and the rules that hold between them as much or as little as they want. Examples of such learning tools include function plotters and dynamic geometry systems. Another example is pictured on the right: the tool Squiggle-M [Fes10] is used to explore relations between finite sets in order to learn about functions, injectivity, and surjectivity.



Aufgabe 2: Beweise oder widerlege mit Hilfe vollständiger Induktion folgende Aussage:

$$\sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6} \quad \text{für alle } n \in \mathbb{N}$$

Induktionsanfang

Lege zuerst die Variable und den Startwert fest, für die der Induktionsanfang durchgeführt wird:

Variable	Startwert
<input type="text" value="n"/>	<input type="text" value="1"/>

Prüfe nun, ob die zu beweisende Aussage für diesen Startwert richtig ist:

Linke Seite: $\sum_{i=1}^1 i^2 = 1^2 = \text{[]}$

Rechte Seite: $\frac{1(1+1)(2 \cdot 1 + 1)}{6} = \text{[]}$

Überlege Dir noch einmal für welche Variable der Induktionsanfang durchgeführt werden muss.

[Tipp](#) [Zurück](#) [Nächstes Problem](#) [Prof. fragen](#)

On the other hand, interactive tools for learning mathematics support the automated corrections of a broad range of typical exercises which allow learners to train and obtain the routine that stabilizes their mastery of the concepts. Trainers of this sort

include the domain-reasoner powered exercises of ActiveMath [Gog09] and many of the cognitive tutors [KAHM97]. Another example is pictured on the left, that of ComIn-M [RZ11]: it allows the learners to apply the classical workflow of proof by induction to proofs about number-theoretical formulae.

Whenever computer tools are used in learning scenarios, the teacher plays a central role: he¹ explains the concepts by using representations and operations that the learner can also find in the learning tools. He invites the learners to use the learning tool: for example, he can indicate that a few exercises of the assignment sheet are to be done using the learning tools which they can find in the learning management systems. These usage incentives are not able to ensure the quality of the usage that enhances what the learners have acquired. Indeed, all sorts of risks appear when learning tools are used:

- An overwhelming cognitive load when introducing the learning tools (caused by the amount of technical details, for example).
- Too steep a learning curve to use the tool (when the students need to concretely apply the mathematical knowledge).
- Choice of exercises that lead to frustration due to unachieved exercises (for mathematical, technical, or other reasons).

These risks are challenges to teachers – the issue is underlined as insufficiently addressed in the report [A11]: *The recurrent difficulties [...] call into question both the design of resources and the processes of their dissemination.*

To assess these risks, teachers must understand the students' usage of the learning tools. But providing detailed logs of the students' actions is not sufficient since the details may be overwhelming. The vast amount of data that could be produced by logging usually prevents the teachers using such a source to deduce any useful information about the students' learning processes.

User Requirements. Approaches and tools are required that allow the efficient analysis and understanding of such log data. Learning analytics is the domain investigating methods and tools to support this.²

¹ In this paper, we shall use the feminine for the learner and the masculine for the teachers even though we mean both genders for both roles.

² Learning analytics is defined by G. Siemens as “the use of intelligent data, learner-produced data, and analysis models to discover information and social connections, and to predict and advise on learning.” in <http://www.elearnspace.org/blog/2010/08/25/what-are-learning-analytics>.

Teachers should be able to capture the overall progression of the learners in a class; being informed on the successes and failures for each exercise and each skilled acquisition aimed at. This has the potential to guide the lecture's content for such adjustments as revising a conceptual error commonly found or demonstrating a manipulation in more detail with the same concepts and representations of the learning tools.

In addition, we consider it important that the teachers complement the automatic assessment capabilities of the learning tools, making it a semi-automatic assessment [BHK⁺11]. Teachers should be able to see the detailed inputs a learner has made and the precise automatic feedback she received when requested to help. In the case of requesting help while using the learning tools, the learners should be able to formulate a request for help linked to the list of events that occurred until they needed assistance so that the teacher can analyze what the learner did and suggest effectively what actions to take next (in both computer-technical and mathematical terms).

In order to realize these objectives, teachers need a logging infrastructure which is the focus of this contribution.

Technical Requirements. The purpose of this research aims at serving teachers in universities to support the use of richly interactive learning tools, typically of client-based applet-like tools which have not been designed with action logging in mind. We aim to insert an architecture for logging into the widespread infrastructure of learning management systems (LMS): those university-central systems that are used for coordination of courses and which each student regularly visits. We aim our development to not require an LMS change; indeed we have often met such a desire to be impossible to satisfy in university wide learning management systems.

Thus, one of the basic technical challenges is that of enabling teachers, who are privileged users of the LMS, to provide their learners with methods to start the learning tools from the LMS so that identified logs are received.

Other technical challenges revolve around the display of relevant information about the learning actions to the teachers in a way that is easily accessible and navigable. The learning tools, the LMSs, and the log views should be web-based and allow the servers to recognize the identity of learners and teachers.

At the Edge of Privacy. A major concern of learning analytics is the set of regulations about the users' privacy. Indeed, the usage of such a monitoring tool may be turned into a powerful watching tool if not used carefully. Moreover, we acknowledge that a part of the students we have met are bothered using a tool where each of the attempted solution paths are always visible.

In comparison to log-views that irreversibly show all steps of the problem solving process, the classroom based usage where teachers and assistants can come and see the current state of the learning tool generally allows the student to cancel (and thus hide) erroneous steps that are irrelevant to a teacher question.

Thus, to support some free choice of disclosure of the students, we set forth the following principles that respect Germany's and EU's laws on privacy:

- The log of a session is only associated to an identifiable person when that individual expressly consents to being identified, i.e. when requesting help.
- The students always have the possibility to opt-out of the log collection.
- The information recorded is transparent to the learner.

These principles do not prevent all sessions of the same learner being grouped together, as long as it remains impossible for a teacher to associate a person with the log view of a session. As we shall describe below, this will be addressed by presenting the learners' *pseudonym*, a barely readable number derived from the name. We acknowledge that teachers would still be able to track regularly by remembering the pseudonym (or by many other means), but we explicitly warn the teachers that such is the start of illicit monitoring.

These principles respect the privacy laws in a same manner as the widespread Twitter or Facebook widgets in web-pages: they do not require a supplementary privacy agreement by the students since the log information that is collected and made available does not contain personal information.

Again similarly to these services, the privacy disclosure is agreed upon when the learners explicitly decide to do so: in the case of such services, this implies registrations, in the case of SMALA, it is when requesting help.

2 State of the Art in Logging User Actions

Logging users' actions can be done in multiple ways; in this section we outline existing approaches that are described in the current research literature. They revolve around two axes: the methods to integrate learning tools in learning management systems and the log-collection and log-view approaches.

2.1 Standards for Integrating Learning Tools

The widespread SCORM packaging standard³ allows makers of learning tools to bundle a sequence of web pages that use an API for communication with the LMS. A more recent derivative of SCORM is Common Cartridge, which extends it with IMS Learning Tools Interoperability IMS-LTI⁴: this specification allows the teacher of a module in an LMS to publish enriched forms of links which carry the authentication.

Both SCORM and Common Cartridge provide basic infrastructures for the web integration of learning tools. However, their logging capabilities in terms of collecting and analysing usage data is very limited. Supported log views typically show tables or counts inside the LMS. For a teacher to be able to evaluate the progress of one learner, he needs to see a less abstract view, more resembling the view the learner had when performing a learning activity.

³ The Shareable Content Object Reference Model standard emerged from ADLnet about 10 y. ago. See <http://www.adlnet.gov/Technologies/scorm/default.aspx>.

⁴ The Learning Tools Interoperability specification is an emerging standard, see <http://www.imsglobal.org/toolsinteroperability2.cfm>.

Multiple other standards have been realized to provide single-sign-on infrastructures: [GRNR09] describes many of these infrastructures and conclude with the proposal of yet another approach to authentication.

2.2 Research Around Log Collections

There are various approaches for collecting logs of user activities and making them available via suitable views. In the following, we will characterize the logging approaches by the level of detail of the collected data, the semantic content that is available, and the analysis capabilities that are offered.

Jacareto: The most detailed approach for log collection is to record each of the user's actions and present these as a *replay* of the learning tool's user interface. This approach has been investigated in a software project called Jacareto [SGZS05]: apart from input events such as mouse clicks in the learning application tool-specific *semantic events* are stored in a recording file. Once the teacher obtains the record file, he can analyze the solution process qualitatively – either by looking at a hierarchical view of the events, or by replaying the events and observing the learning application. However, quantitative analysis of a large number of recordings is not supported, and remote logging is not yet available.

FORMID: The research project FORMID [GC06] aims at applying learning scenarios and view logs based on these scenarios. Each scenario is implemented as a script that specifies a sequence of learning activities to be followed by learners. Additional monitoring facilities enable the teacher to observe learning activities and provide support in real time. In this approach, the semantic content of the logging data is quite rich, and sufficient analysis capabilities are offered for assessing the learners' progression but only within the given scenario.

LOCO-Analyst: The approach of LOCO-Analyst [JGB+08] intends to support the teacher in analyzing learning processes in order to optimize and revise content elements in online learning courses. For this reason, LOCO-Analyst focuses on online learning activities such as text reading and obtaining scores when solving multiple-choice exercises. Based on semantic web technologies, LOCO-Analyst enhances logging data by semantic annotation and provides various analysis services and graphical representations of the collected data. As opposed to Jacareto, the level of detail of the logging data is quite low: it mostly considers high-level events such as page views or successful exercise solutions.

Log Repositories: Learners' activities logs are the basic input for the PSLC DataShop⁵ and for a similar initiative in Kaleidoscope [MMS08]. These initiatives are infrastructures for collecting logs of learning for their massive evaluations for such purposes as data-mining to experimentally measure e-learning theories. These logging repository initiatives are researcher-oriented: they bring together quantities of logs in order to formulate hypotheses about the learners' actions. They are not applicable for teachers, and often do not work on live streams of data.

⁵ For the PSLC DataShop see <https://pslcdatashop.web.cmu.edu/>.

2.3 Conclusions from Literature Review

From this review of the existing literature we conclude that no current standard nor widespread learning management system offers sufficient support for the deployment of logging-enabled learning tools run on the client or even for any interaction-rich learning tools.

We also conclude that the approaches proposed by state-of-the-art research initiatives provide interesting logging features but only few approaches are flexible enough to log semantically-rich events in an adequate level of detail. Customizable views that offer an efficient display of the logging data enhanced by flexible analysis capabilities are still subject to ongoing research.

Figure 1 summarizes how the existing logging approaches can be assigned to the dimensions *amount of detail*, *semantic level*, and *analytics support*. As shown in the diagram, the SMALA architecture that we present in the following chapter aims at high values in all three dimensions, with an intent to cover different amounts of details, and thus represents a new category of logging system.

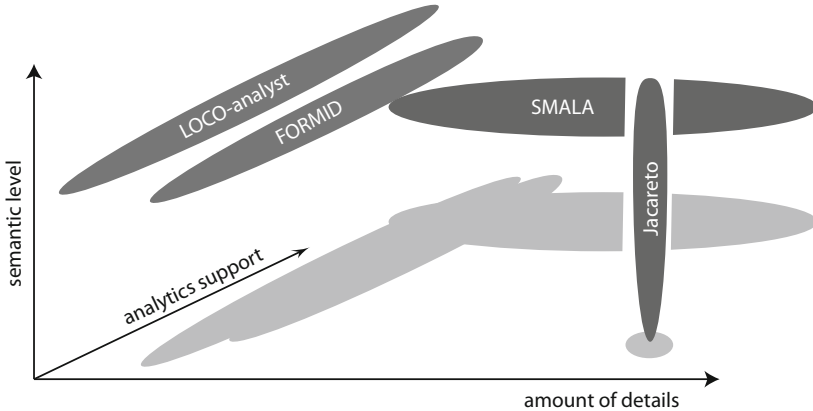


Fig. 1. The various logging systems along three dimensions

3 The SMALA Architecture

The contribution of our paper is an architecture for logging and analyzing learners' activities and solution paths. SMALA, SAIL-M's Architecture for Learning Analytics, responds to the requirements stated above and is realized as a service-oriented web application. Figure 2 depicts the SMALA system architecture where the sequence of components in the creation, deployment, initialization, usage, assessment, logging, and log observation is numbered.

SMALA's core component is the SMALA logging service that is responsible for receiving events from the learning tools and storing them persistently in the logging database. Authorized teachers can retrieve the recorded events from the SMALA web server via suitable views and can analyze the learning activities performed by individuals and groups of learners.

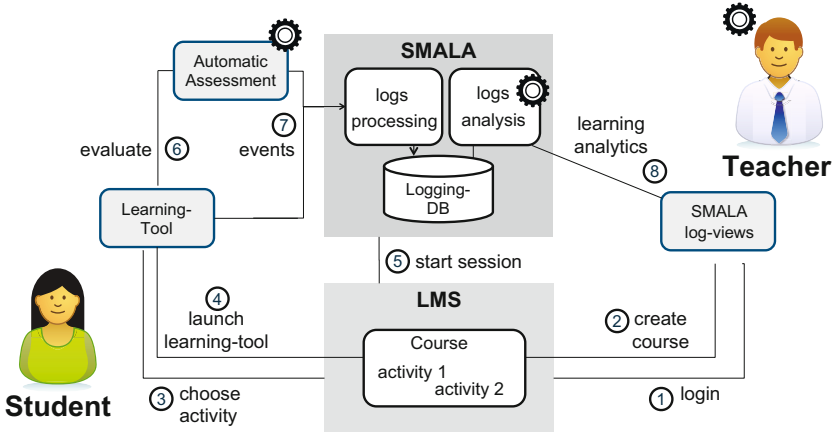


Fig. 2. The SMALA architecture. Gear wheels indicate actors and components with analytical reasoning.

Being a generic logging infrastructure, SMALA offers various interfaces for integrating concrete learning tools into the architecture. These interfaces are related to the authentication, definition, and handling of tool-specific semantic events. The following sections provide a detailed description of these interfaces and their usage.

3.1 Software Components and Their Interactions

The *SMALA server* is a web server which, on the one hand, stores and displays log event streams and, on the other hand, is able to deliver the learning tools to the learners. In order to enable a learning tool to use the SMALA infrastructure, the tool must be registered with one or more *learning activities* in the SMALA environment. This means SMALA knows about the tool and the corresponding activities via XML configuration files and is able to verify that it is allowed to use the SMALA logging service by an application specific key and a whitelist of allowed host URLs. In principle, every learning tool based on web technologies, such as Java applets or AJAX web applications, can be deployed and make use of SMALA's logging facilities.

Typically, SMALA-enabled learning tools are made available to the learners using an LMS (see Section 5) and thus, can be accessed as part of the learning materials available from the LMS course page. In order to use a learning tool, course participants only have to log into the LMS using their usual LMS user account and launch the tool. When starting up the tool, the LMS requests the creation of a new SMALA session for the current LMS user. This session is associated with all subsequent communication requests that occur between the learning tool and the SMALA logging service. As soon as a valid SMALA session is established, the learner can work with the tool in her local web browser. While doing so, all relevant interactions are wrapped as semantic events and

sent to the logging service to store them in the database. Relevant interactions include both the user actions, the assessment results, and the feedback provided by the learning tools as a reaction to the users' activities. Identifying the relevant interactions is the responsibility of the learning tool designers and requires pedagogical expertise and a thorough understanding of the tool's application domain. Based on the SMALA log objects knowledge structure as described in the next section, the tool developers can model custom event objects as needed and add logging functionality to their tools by simply sending these events to the SMALA logging service.

The *SMALA logging service* is implemented as a Java servlet and accepts all logging requests containing a valid SMALA session identifier and a serialized event object. The obtained event object is deserialized by the SMALA logging service and persisted to the SMALA logging database. Event objects have a *type* and are associated with the current *user session* and *learning activity*, i.e. a learning tool configured for a specific LMS course. All learning activities have to be registered with SMALA using XML configuration files and can have so-called *trigger classes* attached to them. If incoming events for these activities match the event type specified by these trigger classes, the triggers are activated and can handle the events appropriately. Currently, our semi-automatic assessment tools make use of this functionality for handling manual feedback events. When the SMALA logging service receives a manual feedback event object, it activates the configured *Send Mail trigger* which sends an email to the designated teacher or tutor, containing the learner's question, her email address, and the associated event data. So as to ensure the pseudonymity of the data, the email address is, then, removed from the event.

Feeding events into the SMALA logging service is completely transparent to the end users of the learning tools. The learners' main focus is on using the learning tools and getting direct feedback from the tool's automatic assessment component. Teachers, however, can make use of SMALA's log views and optional tool-specific summary views on the recorded data (see section 5). These log views and summary views are presented using Java Server Pages (JSP) that query and render the event data from the SMALA logging data base. By allowing the deployment of tool-specific event renderer classes, not only *general event data* such as the timestamp and a textual event description is displayed, but also *tool-specific event data* such as rewritings, error messages or screenshots from the tool's user interface can be shown (see figure 3). In the same way, tool-specific summary views and corresponding analyzer components for processing the semantic event data can be plugged into the SMALA infrastructure and integrated into the standard SMALA log views. The analysis done by these summary views offer the teacher a usable view to support his analysis and decision making.

SMALA offers a flexible *authorization mechanism* to configure roles and access rules per activity: an LMS provenance and a signature enables an authorized *user source*; tutors (that can view log and deployment instructions) are authorized by obtaining identity from external identity providers such as Google and Facebook.

3.2 Log Objects Knowledge Structure

Log-event objects form the basic information entity that describes the users' actions. The semantic-event-based data sets are grouped by *activity* and by *session* and form the starting point of the analytical process.

An activity is SMALA's concept for a learning tool that is offered from within a given course. It is configured with an authorization realm for teachers (a few external accounts) and for students (an integration method into an LMS). Each activity contains sessions which are a stream of log-events for a user.

Log-event objects have been designed with extensibility in mind since their semantics are very tool specific. The basic types include question events, image events, and action events. Common attributes of these basic types include the session-identifier, the timestamp, as well as the exercise name. Based on the exercise name the teacher can identify different parts of the learning activity. Tool makers can refine their types for each of the learning tools, extending the basic types described above. The events can thus include the user's input (such as the OpenMath representations in the case of ComIn-M), the exercise state (such as the formula of the function being plotted in Squiggle-M), or even a URL to reconstruct the exercise state.

The serialization of the events transmitted to the server uses a generic XML format that is able to contain the tool specific logging information in form of key-value pairs, strings, numbers, dates and binary blobs, making the learning tools able to build their custom log event streams out of these datatypes. The events sent from the client are decoded on the server where they are validated and stored in the database using the Java Persistence Architecture.⁶ This form of storage, close to the Java objects nature, allow sophisticated queries to be formulated so that log-views are carefully engineered to be relevant to the analysis of the students' activities for each learning tool.

3.3 Availability

SMALA is publicly available from http://sail-m.de/sail-m/SMALA_en. Its source code (under the Apache Public License), technical documentation, and a link to the production and development servers with some demonstration parts are also linked there.

4 Making the Learning Tools Available

SMALA relies on a continuous identification of the users of the learning tools. The natural way to do so is to integrate the learnings tools in the LMS within the course page where students find their learning materials.

Each activity is established by the SMALA team: it is made by configurations of the roles, deployment explanations, and authorizations. A teacher then simply

⁶ The Java Persistence Architecture is an abstract API for object-relational mappings, see <http://jcp.org/aboutJava/communityprocess/final/jsr317/>.

adds a link to the learning tool to his course page using code fragments suggested on the deployment explanations. As a result, the learning tools can be started one or two clicks away from the main course page in the LMS, and thereby allow teachers to easily promote the tool during the lectures or on assignment sheets.

5 Types of Log Views and Their Usage

Access to the logs starts with a *dashboard view* giving an overview of the recent activity with the learning tools. From there, one can get a list of users and a list of recent sessions. The users are not presented by name, but by their pseudonym, which we expect to be a number impossible to remember by teachers; each of them links to the list of sessions of that user, then to the detailed session-view. From the dashboard, one can also access overview pages giving a global analysis of the class performance.

5.1 Session Views

Sessions are displayed as a chronological sequence of user interactions in all detail. Each of the interactions' types are displayed with a custom representation.

This enables the session log view to display, for example, each input the learner has made, and each fragment of feedback she has received. The purpose of this session view is to document the steps that were performed by the learner in the problem solving process and make it reproducible for the teacher.

This works well with learning tools that are made of dialogs where the learner submits an input and receives feedback. An example is shown in the figure above: it shows a logging session using the learning tool ComIn-M where each of the induction proof constituents are presented. Such detailed log views enable lecturers to reproduce all actions a learner has made during the problem solving process. However, this approach works less well for learning tools oriented towards direct manipulations such as Squiggle-M: log views then become long lists of small actions whose textual rendering is hard to read; e.g. *created point P1 in domain 1, created point P5 in domain 2, ..., linked P1 to P5*).

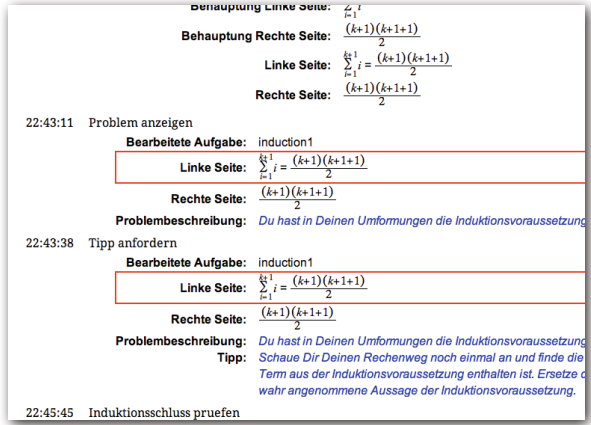


Fig. 3. The log view of a ComInM session

5.2 Requests for Help

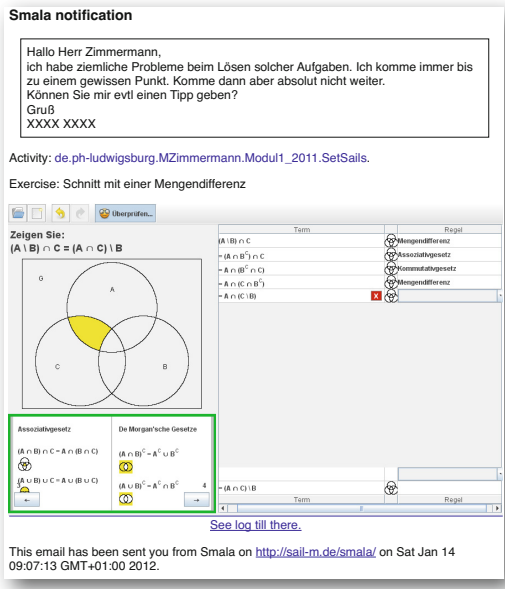


Fig. 4. An actual tutor request email when using SetSails [11] received during the evaluation that a function has been repeatedly used wrongly (therefore being able to guide her accordingly). The semi-automatic feedback paradigm of [11] can be applied fully, enabling teachers to help students use the tool effectively and provide hints that go beyond the automated guidance of the learning tools. An example tutor request email is shown in figure 4.

A special type of interaction realizes the semi-automatic assessment approach [11]: that of a request to the tutor formulated from within the learning tool. This interaction submits a special event to the SMALA server, containing the request of the student and a snapshot representing the current state of the tool (e.g. a screen copy). As soon as the SMALA server receives this event, the request is forwarded to the responsible teacher by email along with a URL to access the session until the moment of submitting the request.

These tutor request e-mails allow the teachers to grasp what the learner did (e.g. observing that a function has been repeatedly used wrongly) therefore being able to guide her accordingly. The semi-automatic feedback paradigm of [11] can be applied fully, enabling teachers to help students use the tool effectively and provide hints that go beyond the automated guidance of the learning tools. An example tutor request email is shown in figure 4.

5.3 Summary Views

The two log views of the previous sections support teachers to help individuals or get an idea of sample solution paths, but they are much less useful when trying to obtain a broader overview of a whole class's advances in the usage of the learning tools.

The most basic summary view is a list of all events of a given type. These allow teachers to find interesting sessions or detect a lack of achievement. Another summary is provided by a dashboard with a timeline graph of the usage activity.

Bearbeitete ComIn-M-Aufgaben:

Anzahl an bearbeiteten Aufgaben: 105
davon richtig gelöst: 97
davon falsch gelöst: 8

Aufgabe	Gesamtanzahl	richtig	falsch	Lösung Prüfen	Tipps	Tutoranfragen
induction1	33	21	12	177	31	3
induction10	5	4	1	21	3	0
induction2	21	16	5	99	24	1
induction3	7	12	5	39	3	1
induction4	6	11	5	46	3	0
induction5	10	14	4	75	14	2
induction6	5	7	2	25	4	0
induction7	6	3	3	23	0	1
induction8	8	0	8	58	10	1
induction9	4	9	5	25	1	0

Fig. 5. A table of progress of the learners

Another summary view is the display of the *success* or *failures* in the usages of the learning tools for each of the exercises that make up a learning tool. An example in figure 5 shows a high involvement in attempting the first few exercises but clear difficulties for the remaining exercises as well as lack of attempts. The teacher could, from such a display, analyze and conclude a lack of understanding of the concepts, lack of engagement, or technical challenges; each of these hypothetical diagnoses could be answered by an in-class demonstration of success in one of the advanced exercises.

6 Evaluation of Learning Tools Using SMALA-Logging

This research has been made in parallel to several evaluations in which both the learning tools' qualities and the logging approach based on SMALA were evaluated.

An early evaluation was run in the summer of 2011 in order to get first feedback from the students and teachers at Karlsruhe University of Education: about forty students brought their laptops in two successive lectures and launched the MoveIt-M learning tool [Fes10] for exploring plane isometries and their composition. This first evaluation showed that obtaining a reasonable network bandwidth for launching the tool (20 MB download) may represent a challenge to other environments. Missing bandwidth may stimulate students to find workarounds such as choosing offline versions of the learning tools. Nonetheless, this first phase showed that transmitting logs is technically easy.

Summative evaluations were run in the winter term of 2011-2012. In the basic mathematics courses of math teachers education three learning tools were deployed and used: Squiggle-M, SetSails [HST11], and ComIn-M. The logs witness the usage of the learning tools by 156 users having run 965 sessions yielding 24655 events. The logs have been complemented by a questionnaire filled out on paper and teacher interviews.

The evaluations have also shown that the transparent integration in which the learning tool is started by the learning management system and the logs are collected under pseudonym works fairly well and is acceptable to the students' privacy requirements. Indeed, only a small portion (less than 2 %) insisted on having no log collection. Teachers that collaborated with us found it acceptable to expand the privacy circle when deploying such learning tools by enabling the learning management system's pages run by learners' browsers to transmit the user-information to the SMALA server. They made this decision because they acknowledged the usefulness and clarity of SMALA's mission.

6.1 Feature Usage Statistics

The central aspect of our evaluation was to measure the relevance of the semi-automatic assessment paradigm. The usage of direct tutor requests from within the learning tools has been low (only 11 requests). Asked why they did not use the tutor request feature, many students responded that the automatic feedback

had been sufficient (26%). Most stated that they preferred asking peers (36%) or tutors (26%) directly (thus waiting till they had the chance to ask a question personally), while a few responded that no help was necessary (11%), or that formulating the question turned to be hard (15%).

Although only few help requests were formulated (such as the one in figure 4), the requests that were actually submitted to the teachers could be answered: the detailed log views of the sessions have been sufficiently complete for them to answer questions with a precise hint that set the students on the right path. For this central workflow, the evaluation showed a successful setup.

A teacher who had seen the logs of ComIn-M commended the *screenshots of the users' input*. In fact, the logs do not contain screenshots, but MathML representations of the user's input. This comment shows that a faithful graphical display of the entire mathematical formulas is important to understand the students' solutions.

The usage of the logging feature for each of the teachers has been almost limited to responding to individual requests. When interviewed at the end of the project, almost all teachers said they would consider using SMALA logging views more intensively if richer summary views were offered. Indeed, we focused on the log views to support individual requests, expecting overviews to be obtained by sampling a few sessions. As a result such summary views as the table in figure 5 were implemented late in the development cycle. Our interviews also indicated that the separation of statistics *by exercise* seems to be an important basis of most log views.

Analyzing the web-server logs (1813 page views) showed that the students made use of the *myLog* links displayed aside of each tool launch. This feature is offered to support the transparency requirement. It allows each student to review her activities in the same format in which a teacher would see it.

6.2 Technical Challenges

The development of SMALA towards its sustained usage in day-to-day classrooms has been iterative, based on the feedback of the teachers and the students. It has faced the following technical challenges:

Genericity. The SMALA server was designed to serve both web-applications and rich-client applications and it succeeded in doing so. Both ComIn-M (a web-server that sent all its logs to SMALA) and Java Applets (SquiggleM, MoveItM, and SetSails) could start and send identified logs to the SMALA server. Although the learning tools send log-events of different types through different connection methods, the resulting log displays are delivered in a unified user interface.

Scalability. The choice of persistence layers such as the Java Persistence Architecture has been an important key to ensure the scalability to several thousands of log entries. For some display methods, large amounts of events still have to be inspected and it is still a matter of a few seconds (e.g. to display thousands of log entries). However, SMALA has sometimes been at the edge of performance even

though it only handled a handful of courses. For example, listing long sessions or listing all uploaded screenshots takes several minutes during which the data is extracted from the database and converted for display. Summary views, in particular, need to strictly enforce the rule that their results are fetched using statistical queries to the database, which the tool developers enter using the JPQL language instead of iterating through numerous events.

An aspect at which the persistence libraries have shown to be too fragile is that of the evolution of the database schema of the log events. While creating an SQL schema based on the Java properties is transparent and effective (including support in IDEs to formulate queries), the persistence architecture offers almost no support to subsequently upgrade the SQL schema.

A modification such as raising the maximum size of a property (hence the size of an SQL column) needs either to be done manually in SQL or by an externalisation followed by a rebuild of the database. We ran the rebuild process at almost every deployment, the last taking about three hours. This process is, however, risky, and only thorough testing revealed that some information was lost in the process. The web nature of the log views, accessible by simple URLs allowed an easy test infrastructure: a simple web-page described each piece of information expected at each log-viewer URL. This allowed us to ensure that there are no unintended effects of the new versions' deployment.

7 Conclusion

In this paper we have described an approach to convey to practicing teachers what their students have been doing with learning tools. The approach relies on the usage of a software whose role is both to make available the learning tool in the learning management system and to display log views representing the learner's activity. This approach turns the automatic feedback of the assessment tools into more intelligent tools which can exploit the teacher's knowledge. The implemented toolset conveys broad enough information to watch the learners' overall progress, and to respond to individual queries.

Below we present research questions that this approach opens.

7.1 Open Questions

The involvement of the teachers in the usage of the learning tools remains a broad area of research which this paper contributes to. It is a subtle mix of mathematical competency, technical skills, and pedagogical sense – all of which are addressed by a teacher support tool. It is bound to the teachers' representations of the mathematical knowledge that students have in mind, the meaning they attach behind them and behind the operations with them.

The *instrumental orchestration* [DT08] is a model of the role of the teacher who uses the instruments (the tools) to let himself and the students (the orchestra) play the mathematical music. It mostly studies the usage of the learning tools in class, while students often use learning tools in the comfort of their homes:

the learning tools are there, precisely, to let the learners deepen or discover at their own pace, away from the tight rhythm of the lectures.

The research of this paper contributes to letting the teacher perceive this activity. In this paper we have proposed a few solutions relying on both detailed and summary views. Does it imply a high level of diagnostics? The survey [ZS06] indicates that higher level indicators such as the learners' mastery level and misconceptions are much desired. We note, however, that they would only be useful if they can be trusted by teachers to be complete and precise; we are not sure that this is easily achievable, and indeed the paper highlights this difficulty in its conclusion. Sticking to more concrete events, such as *typical feedback types* might be easier to implement, easier to represent to the teacher, directly linkable to steps in session actions, and thus much more trustable.

We have introduced SMALA in relatively traditional didactical settings in which the learning tools are mostly used for the purposes of exercising. This is an important aspect but other didactical setups can leverage a logging architecture.

Among didactical setups is the exploitation of the *own log* feature beyond the mere transparency requirement. Displaying the individual logs of a selected learner, this feature can be used by the student as a personal log to be reminded of the competencies she has already acquired, much similar to the learning-log approach. What needs to be achieved to encourage and make possible such practice? Should links be made from a personal diary into concrete sessions?

As another example of a different didactical setup, Erica Melis suggested to display the logs of past uses of the learning tool overlaid on the *behaviour graph* of the Cognitive Tutor Authoring Tools [AMSK09] and to use this in coordination with the learning tools. This approach would constitute a discussion tool that allows the teacher to explain the possible avenues to solve the exercise, changing the state of the exercise and discussing what was correct or what was wrong. An approach and architecture such as SMALA makes such a tool possible.

Acknowledgements. This research has been partly funded by the Ministry of Education and Research of Germany in the SAiL-M project. We thank our student assistants Torsten Kammer and Michael Binder for their active support.

References

- AI11. Michèle Artigue and International Group of Experts on Science and Mathematics Education Policies. Les défis de l'enseignement des mathématiques dans l'éducation de base. Number 191776 - UNESCO Archive, IBE Geneva (2011)
- AMSK09. Aleven, V., McLaren, B., Sewall, J., Koedinger, K.: A new paradigm for intelligent tutoring systems: Example-tracing tutors. *International Journal of Artificial Intelligence in Education* 20 (2009)
- BHK⁺11. Bescherer, C., Herding, D., Kortenkamp, U., Müller, W., Zimmermann, M.: E-learning tools with intelligent assessment and feedback. In: Graf, S., Lin, F., Kinshuk, McGreal, R. (eds.) *Intelligent and Adaptive Learning Systems: Technology Enhanced Support for Learners and Teachers*. IGI Global, Addison Wesley (2011) (in press)

- DT08. Drijvers, P., Trouche, L.: From artifacts to instruments: a theoretical framework behind the orchestra metaphor. In: Heid, K., Blume, G. (eds.) *Research on Technology and the Teaching and Learning of Mathematics, Information Age*, Charlotte, NC, USA, pp. 363–392 (2008)
- Fes10. Fest, A.: Creating interactive user feedback in DGS using scripting interfaces. *Acta Didactica Napocensia* 3(2) (2010)
- GC06. Guéraud, V., Cagnat, J.-M.: Automatic Semantic Activity Monitoring of Distance Learners Guided by Pedagogical Scenarios. In: Nejdil, W., Tochtermann, K. (eds.) *EC-TEL 2006. LNCS*, vol. 4227, pp. 476–481. Springer, Heidelberg (2006)
- Gog09. Gogvadze, G.: Representation for Interactive Exercises. In: Carette, J., Dixon, L., Coen, C.S., Watt, S.M. (eds.) *MKM 2009, Held as Part of CICM 2009. LNCS*, vol. 5625, pp. 294–309. Springer, Heidelberg (2009)
- GRNR09. Fontenla González, J., Rodríguez, M., Nistal, M., Anido Rifón, L.: Reverse oauth: A solution to achieve delegated authorizations in single sign-on e-learning systems. *Computers and Security* 28(8), 843–856 (2009)
- HS11. Herding, D., Schroeder, U.: Using capture & replay for semi-automatic assessment. In: Whitelock, D., Warburton, W., Wills, G., Gilbert, L. (eds.) *CAA 2011 International Conference* (2011)
- JGB⁺08. Jovanovic, J., Gasevic, D., Brooks, C., Devedzic, V., Hatala, M., Eap, T., Richards, G.: LOCO-Analyst: semantic web technologies in learning content usage analysis. *International Journal of Continuing Engineering Education and Life Long Learning* 18(1), 54–76 (2008)
- KAHM97. Koedinger, K., Anderson, J., Hadley, W., Mark, M.: Intelligent tutoring goes to school in the big city. *IJAIED* 8(1) (1997)
- MMS08. Melis, E., McLaren, B., Solomon, S.: Towards Accessing Disparate Educational Data in a Single, Unified Manner. In: Dillenbourg, P., Specht, M. (eds.) *EC-TEL 2008. LNCS*, vol. 5192, pp. 280–283. Springer, Heidelberg (2008)
- RZ11. Rebholz, S., Zimmermann, M.: Applying computer-aided intelligent assessment in the context of mathematical induction. In: *eLearning Baltics 2011*. Fraunhofer Verlag, Stuttgart (2011)
- SGZS05. Spannagel, C., Gläser-Zikuda, M., Schroeder, U.: Application of qualitative content analysis in user-program interaction research. *Forum Qualitative Sozialforschung / Forum: Qualitative Social Research* 6(2) (2005)
- ZS06. Zinn, C., Scheuer, O.: Getting to Know Your Student in Distance Learning Contexts. In: Nejdil, W., Tochtermann, K. (eds.) *EC-TEL 2006. LNCS*, vol. 4227, pp. 437–451. Springer, Heidelberg (2006)

Towards Understanding Triangle Construction Problems*

Vesna Marinković and Predrag Janičić

Faculty of Mathematics, University of Belgrade, Serbia

Abstract. Straightedge and compass construction problems are one of the oldest and most challenging problems in elementary mathematics. The central challenge, for a human or for a computer program, in solving construction problems is a huge search space. In this paper we analyze one family of triangle construction problems, aiming at detecting a small core of the underlying geometry knowledge. The analysis leads to a small set of needed definitions, lemmas and primitive construction steps, and consequently, to a simple algorithm for automated solving of problems from this family. The same approach can be applied to other families of construction problems.

Keywords: Triangle construction problems, automated deduction in geometry.

1 Introduction

Triangle construction problems (in Euclidean plane) are problems in which one has to construct, using straightedge¹ and compass² a triangle that meets given (usually three) constraints [10, 24, 26]. The central problem, for a human or for a computer program, in solving triangle construction problems is a huge search space: primitive construction steps can be applied in a number of ways, exploding further along the construction. Consider, as an illustration, the following simple problem: *given the points A , B , and G , construct a triangle ABC such that G is its barycenter*. One possible solution is: construct the midpoint M_c of the segment AB and then construct a point C such that $\overrightarrow{M_cG}/\overrightarrow{M_cC} = 1/3$ (Figure 1). The solution is very simple and intuitive. However, if one wants to describe a systematic (e.g., automatic) way for reaching this solution, he/she should consider a wide range of possibilities. For instance, after constructing the point M_c ,

* This work was partially supported by the Serbian Ministry of Science grant 174021 and by Swiss National Science Foundation grant SCOPES IZ73Z0_127979/1.

¹ The notion of “straightedge” is weaker than “ruler”, as ruler is assumed to have markings which could be used to make measurements. Geometry constructions typically require use of straightedge and compass, not of ruler and compass.

² By compass, we mean *collapsible compass*. In contrast to *rigid compass*, one cannot use collapsible compass to “hold” the length while moving one point of the compass to another point. One can only use it to hold the radius while one point of the compass is fixed [3].

one might consider constructing midpoints of the segments AG , BG , or even of the segments AM_c , BM_c , GM_c , then midpoints of segments with endpoints among these points, etc. Instead of the step that introduces the point C such that $\overrightarrow{M_cG}/\overrightarrow{M_cC} = 1/3$ one may (unnecessarily) consider introducing a point X such that $\overrightarrow{AG}/\overrightarrow{AX} = 1/3$ or $\overrightarrow{BG}/\overrightarrow{BX} = 1/3$. Also, instead of the step that introduces the point C such that $\overrightarrow{M_cG}/\overrightarrow{M_cC} = 1/3$ one may consider introducing a point X such that $\overrightarrow{M_cG}/\overrightarrow{M_cX} = k$, where $k \neq 1/3$, etc. Therefore, this trivial example shows that any systematic solving of construction problems can face a huge search space even if only two high-level constructions steps that are really needed are considered. Additional problem in solving construction problems makes the fact that some of them are unsolvable (which can be proved by an algebraic argument), including, for instance, three antiquity geometric problems: circle squaring, cube duplication, angle trisection [31]. Although the problem of constructibility (using straightedge and compass) of a figure that can be specified by algebraic equations with coefficients in \mathbf{Q} is decidable [14, 17, 23], there are no simple and efficient, “one-button” implemented decision procedures so, typically, proofs of insolubility of construction problems are made *ad-hoc* and not derived by uniform algorithms.

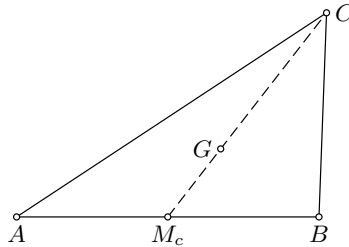


Fig. 1. Construction of a triangle ABC given its vertices A , B and the barycenter G

Construction problems have been studied, since ancient Greeks, for centuries and represent a challenging area even for experienced mathematicians. Since early twentieth century, “triangle geometry”, including triangle construction problems, has not been considered a premier research field [8]. However, construction problems kept their role on all levels of mathematical education, almost in the same form for more than two and a half millenia, which make them probably the problems used most constantly throughout the history of mathematical education. Since the late twentieth century, geometry constructions are again a subject of research, but this time mainly meta-mathematical. There are two main lines of this work:

- **Research in Axiomatic Foundations of Geometry Constructions and Foundational Issues.** According to Pambuccian and his survey of axiomatizing geometric constructions, surprisingly, it is only in 1968 that geometric constructions became part of axiomatizations of geometry [28]. In constructive geometry axiomatizations, following ancient understanding,

axioms are quantifier-free and involve only operation symbols (reflecting construction steps) and individual constants, in contrast to the Hilbert-style approach with relation symbols and where axioms are not universal statements. One such axiomatic theory of constructive geometry — **ECG** (“Elementary Constructive Geometry”) was recently proposed by Beeson [3]. Constructive axiomatizations bring an alternative approach in geometry foundations, but they do not bring a substantial advantage to the Hilbert style when it comes to solving concrete construction problems.

- **Research in Developing Algorithms for Solving Construction Problems.** There are several approaches for automated solving of construction problems [13, 15, 16, 29]. However, most, if not all of them, focus on search procedures and do not focus on finding a small portion of geometry knowledge (axioms and lemmas) that are underlying the constructions (although, naturally, all approaches have strict lists of available primitive construction steps). Earlier attempts at (manual) systematization of triangle construction problems also didn’t provide small and clear, possibly minimal in some sense, lists of needed underlying geometry knowledge [11, 12, 24].

We find that it is important to locate, understand and systematize the knowledge relevant for solving construction problems or their subclasses. That should be useful for teachers, students and mathematical knowledge base generally. Also, such understanding should lead to a system that automatically solves these kinds of problems (and should be useful in education).

In this work we focus on one family of triangle construction problems and try to derive an algorithm for automated solving of problems based on a small portion of underlying geometry knowledge. Our analyses lead us to a small set of definitions, lemmas and construction rules needed for solving most of the solvable problems of this family. The same approach can be applied to other sorts of triangle construction problems and, more generally, to other sorts of construction problems. The approach, leading to a compact representation of the underlying geometry knowledge, can be seen not only as an algorithm for automated solving of triangle construction problems but also as a work in geometry knowledge management, providing a compact representation for a large sets of construction problems, currently not available in the literature.

2 Constructions by Straightedge and Compass

There are several closely related definitions of a notion of constructions by straightedge and compass [3, 9, 31]. By a straightedge-and-compass construction we will mean a sequence of the following primitive (or *elementary*) steps:

- construct an arbitrary point (possibly distinct from some given points);
- construct (with *ruler*) the line passing through two given distinct points;
- construct (with *compass*) the circle centered at some point passing through another point;
- construct an intersection (if it exists) of two circles, two lines, or a line and a circle.

In describing geometrical constructions, both primitive and compound constructions can be used. A straightedge-and-compass construction problem is a problem in which one has to provide a straightedge-and-compass construction such that the constructed objects meet given conditions. A solution of a geometrical construction problem traditionally includes the following four phases/components [1, 10, 18, 24]:

Analysis: In analysis one typically starts from the assumption that a certain geometrical object satisfies the specification Γ and proves that properties Λ enabling the construction also hold.

Construction: In this phase, straightedge-and-compass construction based on the analysis (i.e., on the properties Λ which are proved within it) has to be provided.

Proof: In this phase, it has to be proved that the provided straightedge-and-compass construction meets the given specification, i.e., the conditions Γ .

Discussion: In the discussion, it is considered how many possible solutions to the problem there exist and under which conditions.

3 Wernick's Problems

In 1982, Wernick presented a list of triangle construction problems [34]. In each problem, a task is to construct a triangle from three located points selected from the following set of 16 characteristic points:

- A, B, C, O : three vertices and circumcenter;
- M_a, M_b, M_c, G : the side midpoints and barycenter (centroid);
- H_a, H_b, H_c, H : three feet of altitudes and orthocenter;
- T_a, T_b, T_c, I : three feet of the internal angles bisectors, and incenter;

There are 560 triples of the above points, but Wernick's list consists only of 139 significantly different non-trivial problems. The triple (A, B, C) is trivial and, for instance, the problems (A, B, M_a) , (A, B, M_b) , (B, C, M_b) , (B, C, M_c) , (A, C, M_a) , and (A, C, M_c) are considered to be symmetric (i.e., analogous). Some triples are redundant (e.g., (A, B, M_c) — given points A and B , the point M_c is uniquely determined, so it is redundant in (A, B, M_c)), but are still listed and marked **R** in Wernick's list. Some triples are constrained by specific conditions, for instance, in (A, B, O) the point O has to belong to the perpendicular bisector of AB (and in that case there are infinitely many solutions). In these problems, the locus restriction gives either infinitely many or no solutions. These problems are marked **L** in Wernick's list. There are 113 problems that do not belong to the groups marked **R** and **L**. Problems that can be solved by straightedge and ruler are marked **S** and problems that cannot be solved by straightedge and ruler are marked **U**.

In the original list, the problem 102 was erroneously marked **S** instead of **L** [27]. Wernick's list left 41 problem unresolved/unclassified, but the update from 1996 [27] left only 20 of them. In the meanwhile, the problems 90, 109, 110, 111 [30], and 138 [33] were proved to be unsolvable. Some of the problems

Table 1. Wernick's problems and their current status

1. A, B, O	L	36. A, M_b, T_c	S	71. O, G, H	R	106. M_a, H_b, T_c	U [27]
2. A, B, M_a	S	37. A, M_b, I	S	72. O, G, T_a	U [27]	107. M_a, H_b, I	U [27]
3. A, B, M_c	R	38. A, G, H_a	L	73. O, G, I	U [27]	108. M_a, H, T_a	U [27]
4. A, B, G	S	39. A, G, H_b	S	74. O, H_a, H_b	U [27]	109. M_a, H, T_b	U [30]
5. A, B, H_a	L	40. A, G, H	S	75. O, H_a, H	S	110. M_a, H, I	U [30]
6. A, B, H_c	L	41. A, G, T_a	S	76. O, H_a, T_a	S	111. M_a, T_a, T_b	U [30]
7. A, B, H	S	42. A, G, T_b	U [27]	77. O, H_a, T_b		112. M_a, T_a, I	S
8. A, B, T_a	S	43. A, G, I	S [27]	78. O, H_a, I		113. M_a, T_b, T_c	
9. A, B, T_c	L	44. A, H_a, H_b	S	79. O, H, T_a	U [27]	114. M_a, T_b, I	U [27]
10. A, B, I	S	45. A, H_a, H	L	80. O, H, I	U [27]	115. G, H_a, H_b	U [27]
11. A, O, M_a	S	46. A, H_a, T_a	L	81. O, T_a, T_b		116. G, H_a, H	S
12. A, O, M_b	L	47. A, H_a, T_b	S	82. O, T_a, I	S [27]	117. G, H_a, T_a	S
13. A, O, G	S	48. A, H_a, I	S	83. M_a, M_b, M_c	S	118. G, H_a, T_b	
14. A, O, H_a	S	49. A, H_b, H_c	S	84. M_a, M_b, G	S	119. G, H_a, I	
15. A, O, H_b	S	50. A, H_b, H	L	85. M_a, M_b, H_a	S	120. G, H, T_a	U [27]
16. A, O, H	S	51. A, H_b, T_a	S	86. M_a, M_b, H_c	S	121. G, H, I	U [27]
17. A, O, T_a	S	52. A, H_b, T_b	L	87. M_a, M_b, H	S [27]	122. G, T_a, T_b	
18. A, O, T_b	S	53. A, H_b, T_c	S	88. M_a, M_b, T_a	U [27]	123. G, T_a, I	
19. A, O, I	S	54. A, H_b, I	S	89. M_a, M_b, T_c	U [27]	124. H_a, H_b, H_c	S
20. A, M_a, M_b	S	55. A, H, T_a	S	90. M_a, M_b, I	U [30]	125. H_a, H_b, H	S
21. A, M_a, G	R	56. A, H, T_b	U [27]	91. M_a, G, H_a	L	126. H_a, H_b, T_a	S
22. A, M_a, H_a	L	57. A, H, I	S [27]	92. M_a, G, H_b	S	127. H_a, H_b, T_c	
23. A, M_a, H_b	S	58. A, T_a, T_b	S [27]	93. M_a, G, H	S	128. H_a, H_b, I	
24. A, M_a, H	S	59. A, T_a, I	L	94. M_a, G, T_a	S	129. H_a, H, T_a	L
25. A, M_a, T_a	S	60. A, T_b, T_c	S	95. M_a, G, T_b	U [27]	130. H_a, H, T_b	U [27]
26. A, M_a, T_b	U [27]	61. A, T_b, I	S	96. M_a, G, I	S [27]	131. H_a, H, I	S [27]
27. A, M_a, I	S [27]	62. O, M_a, M_b	S	97. M_a, H_a, H_b	S	132. H_a, T_a, T_b	
28. A, M_b, M_c	S	63. O, M_a, G	S	98. M_a, H_a, H	L	133. H_a, T_a, I	S
29. A, M_b, G	S	64. O, M_a, H_a	L	99. M_a, H_a, T_a	L	134. H_a, T_b, T_c	
30. A, M_b, H_a	L	65. O, M_a, H_b	S	100. M_a, H_a, T_b	U [27]	135. H_a, T_b, I	
31. A, M_b, H_b	L	66. O, M_a, H	S	101. M_a, H_a, I	S	136. H, T_a, T_b	
32. A, M_b, H_c	L	67. O, M_a, T_a	L	102. M_a, H_b, H_c	L	137. H, T_a, I	
33. A, M_b, H	S	68. O, M_a, T_b	U [27]	103. M_a, H_b, H	S	138. T_a, T_b, T_c	U [33]
34. A, M_b, T_a	S	69. O, M_a, I	S	104. M_a, H_b, T_a	S	139. T_a, T_b, I	S
35. A, M_b, T_b	L	70. O, G, H_a	S	105. M_a, H_b, T_b	S		

were additionally considered for simpler solutions, like the problem 43 [2, 7], the problem 57 [35], or the problem 58 [6, 30]. Of course, many of the problems from Wernick's list were considered and solved along the centuries without the context of this list. The current status (to the best of our knowledge) of the problems from Wernick's list is given in Table 1: there are 72 **S** problems, 16 **U** problems, 3 **R** problems, and 23 **L** problems. Solutions for 59 solvable problems can be found on the Internet [30].

An extended list, involving four additional points (E_a, E_b, E_c — three Euler points, which are the midpoints between the vertices and the orthocenter and N — the center of the nine-point circle) was presented and partly solved by Connelly [5]. There are also other variations of Wernick's list, for instance, the list of problems to be solved given three out of the following 18 elements: sides a, b, c ; angles α, β, γ ; altitudes h_a, h_b, h_c ; medians m_a, m_b, m_c ; angle bisectors t_a, t_b, t_c ; circumradius R ; inradius r ; and semiperimeter s . There are 186 significantly different non-trivial problems, and it was reported that (using Wernick's notation) 3 belong to the **R** group, 128 belong to the **S** group, 27 belong to the **U** group, while the status of the remaining ones was unknown [4]. In addition to the above elements, radii of external incircles r_a, r_b, r_c and the triangle

area S can be also considered, leading to the list of 22 elements and the total of 1540 triples. Lopes presented solutions to 371 non-symmetric problems of this type [24] and Fursenko considered the list of (both solvable and unsolvable) 350 problems of this type [11, 12].

4 Underlying Geometry Knowledge

Consider again the problem from Section 1 (it is problem 4 from Wernick's list). One solution is as follows: *construct the midpoint M_c of the segment AB and then construct a point C such that $\overrightarrow{M_cG}/\overrightarrow{M_cC} = 1/3$* . Notice that this solution implicitly or explicitly uses the following:

1. M_c is the side midpoint of AB (definition of M_c);
2. G is the barycenter of ABC (definition of G);
3. it holds that $\overrightarrow{M_cG}/\overrightarrow{M_cC} = 1/3$ (lemma);
4. it is possible to construct the midpoint of a line segment;
5. given points X and Y , it is possible to construct a point Z , such that $\overrightarrow{XY}/\overrightarrow{XZ} = 1/3$.

However, the nature of the above properties is typically not stressed within solutions of construction problems and the distinctions are assumed. Of course, given a proper proof that a construction meets the specification, this does not really affect the quality of the construction, but influences the meta-level understanding of the domain and solving techniques that it admits. Following our analyses of Wernick's list, we insist on a clear separation of concepts in the process of solving construction problems: separation into definitions, lemmas (geometry properties), and construction primitives. This separation will be also critical for automating the solving process.

Our analyses of available solutions of Wernick's problems³ led to the list of 67 high-level construction rules, many of which were based on complex geometry properties and complex compound constructions. We implemented a simple forward chaining algorithms using these rules and it was able to solve each of solvable problems within 1s. Hence, the search over this list of rules is not problematic — what is problematic is how to represent the underlying geometry knowledge and derive this list. Hence, our next goal was to derive high-level construction steps from the (small) set of definitions, lemmas and construction primitives. For instance, from the following:

- it holds that $\overrightarrow{M_cG} = 1/3\overrightarrow{M_cC}$ (lemma);
- given points X , Y and U , and a rational number r , it is possible to construct a point Z such that: $\overrightarrow{XY}/\overrightarrow{UZ} = r$ (construction primitive; Figure 2);

we should be able to derive:

- given M_c and G , it is possible to construct C .

³ In addition to 59 solutions available from the Internet [30], we solved 6 problems, which leaves us with 7 solvable problems with no solutions.

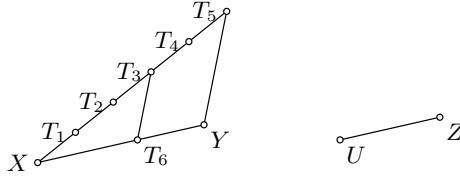


Fig. 2. Illustration for the construction: given points X , Y and U , and a rational number r , it is possible to construct a point Z such that: $\overrightarrow{XY}/\overrightarrow{UZ} = r$ (for $r = 5/3$)

After a careful study, we came to a relatively small list of geometry properties and primitive constructions needed. In the following, we list all definitions, geometry properties, and primitive constructions needed for solving most of the problems from Wernick's list that are currently known to be solvable. The following notation will be used: XY denotes the line passing through the distinct points X and Y ; \overline{XY} denotes the segment with endpoints X and Y ; \overrightarrow{XY} denotes the vector with endpoints X and Y ; $k(X, Y)$ denotes the circle with center X that passes through Y ; $\mathcal{H}(X, Y; Z, U)$ denotes that the pair of points X, Y is harmonically conjugate with the pair of points Z, U (i.e., $\overrightarrow{XU}/\overrightarrow{UY} = -\overrightarrow{XZ}/\overrightarrow{ZY}$); $s_p(X)$ denotes the image of X in line reflection with respect to a line p ; *homothety* $_{X,r}(Y)$ denotes the image of Y in homothety with respect to a point X and a coefficient r .

4.1 Definitions

Before listing the definitions used, we stress that we find the standard definition of the barycenter (*the barycenter of a triangle is the intersection of the medians*) and the like — inadequate. Namely, this sort of definitions hides a non-trivial property that all three medians (the lines joining each vertex with the midpoint of the opposite side) do intersect in one point. Our, constructive version of the definition of the barycenter says that the barycenter G of a triangle ABC is the intersection of two medians AM_a and BM_b (it follows directly from Pasch's axiom that this intersection exists). Several of the definitions given below are formulated in this spirit. Notice that, following this approach, in contrast to the Wernick's criterion, for instance, the problems (A, B, G) and (A, C, G) are not symmetrical (but we do not revise Wernick's list).

For a triangle ABC we denote by (along Wernick's notation; Figure 3):

1. M_a, M_b, M_c (the side midpoints): points such that $\overrightarrow{BM_a}/\overrightarrow{BC} = 1/2$, $\overrightarrow{CM_b}/\overrightarrow{CA} = 1/2$, $\overrightarrow{AM_c}/\overrightarrow{AB} = 1/2$;
2. O (circumcenter): the intersection of lines perpendicular at M_a and M_b on BC and AC ;
3. G (barycenter): the intersection of AM_a and BM_b ;
4. H_a, H_b, H_c : intersections of the side perpendiculars with the opposite sides;
5. H (orthocenter): the intersection of AH_a and BH_b ;
6. T_a, T_b, T_c : intersections of the internal angles bisectors with the opposite sides;

7. I (incenter): the intersection of AT_a and BT_b ;
8. T'_a, T'_b, T'_c : intersections of the external angles bisectors with the opposite sides;
9. $H'_{BC}, H'_{AC}, H'_{AB}$: images of H in reflections over lines BC, AC and AB ;
10. P_a, P_b, P_c : feet from I on BC, AC and AB ;
11. N_a, N_b, N_c : intersections of OM_a and AT_a, OM_b and BT_b, OM_c and CT_c .

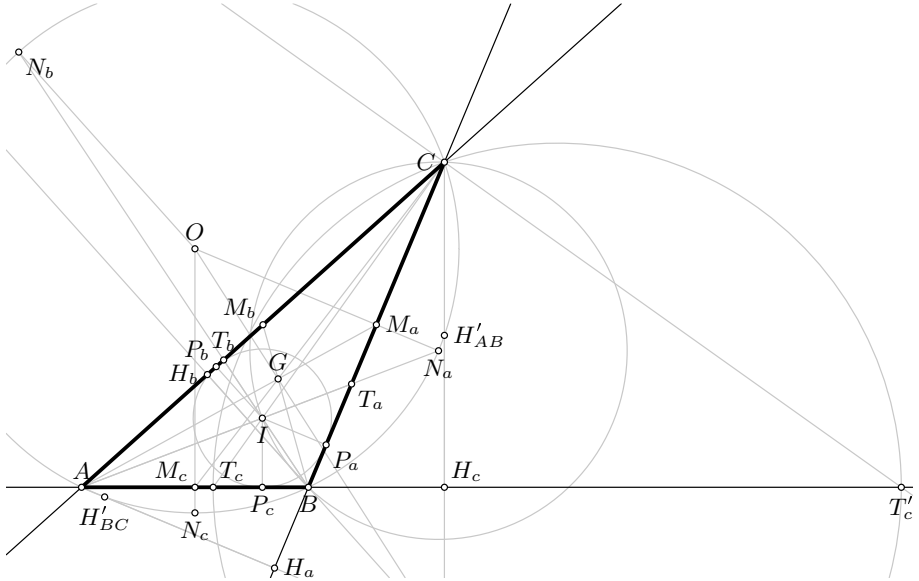


Fig. 3. Points used in solutions to Wernick's problems

4.2 Lemmas

For a triangle ABC it holds that (Figure 3):

1. O is on the line perpendicular at M_c on AB ;
2. G is on CM_c ;
3. H is on CH_c ;
4. I is on CT_c ;
5. B and C are on $k(O, A)$;
6. P_b and P_c are on $k(I, P_a)$;
7. $\overrightarrow{AG}/\overrightarrow{AM_a} = 2/3, \overrightarrow{BG}/\overrightarrow{BM_b} = 2/3, \overrightarrow{CG}/\overrightarrow{CM_c} = 2/3$;
8. $\overrightarrow{M_bM_a}/\overrightarrow{AB} = 1/2, \overrightarrow{M_cM_b}/\overrightarrow{BC} = 1/2, \overrightarrow{M_cM_a}/\overrightarrow{AC} = 1/2$;
9. $\overrightarrow{HG}/\overrightarrow{HO} = 2/3$;
10. $\overrightarrow{M_aO}/\overrightarrow{HA} = 1/2, \overrightarrow{M_bO}/\overrightarrow{HB} = 1/2, \overrightarrow{M_cO}/\overrightarrow{HC} = 1/2$;
11. AB, BC, CA touch $k(I, P_a)$;

12. N_a, N_b, N_c are on $k(O, A)$;
13. $H'_{BC}, H'_{AC}, H'_{AB}$ are on $k(O, A)$;
14. C, H_b, H_c are on $k(M_a, B)$; A, H_a, H_c are on $k(M_b, C)$; B, H_a, H_b are on $k(M_c, A)$;
15. B, I are on $k(N_a, C)$; C, I are on $k(N_b, A)$; A, I are on $k(N_c, B)$;
16. AH, BH, CH are internal angles bisectors of the triangle $H_aH_bH_c$;
17. $\mathcal{H}(B, C; T_a, T'_a), \mathcal{H}(A, C; T_b, T'_b), \mathcal{H}(A, B; T_c, T'_c)$;
18. A is on the circle with diameter $T_aT'_a$; B is on the circle with diameter $T_bT'_b$; C is on the circle with diameter $T_cT'_c$;
19. $\angle T_cIT_b = \angle BAC/2 + \pi/2$; $\angle T_bIT_a = \angle ACB/2 + \pi/2$; $\angle T_aIT_c = \angle CBA/2 + \pi/2$;
20. The center of a circle is on the side bisector of its arbitrary arc;
21. If the points X and Y are on a line p , so is their midpoint;
22. If $\overrightarrow{XY}/\overrightarrow{ZW} = r$ then $\overrightarrow{YX}/\overrightarrow{WZ} = r$;
23. If $\overrightarrow{XY}/\overrightarrow{ZW} = r$ then $\overrightarrow{ZW}/\overrightarrow{XY} = 1/r$;
24. If $\overrightarrow{XY}/\overrightarrow{ZW} = r$ then $\overrightarrow{WZ}/\overrightarrow{YX} = 1/r$;
25. If $\overrightarrow{XY}/\overrightarrow{XW} = r$ then $\overrightarrow{WY}/\overrightarrow{WX} = 1 - r$;
26. If $\mathcal{H}(X, Y; Z, W)$ then $\mathcal{H}(Y, X; W, Z)$;
27. If $\mathcal{H}(X, Y; Z, W)$ then $\mathcal{H}(Z, W; X, Y)$;
28. If $\mathcal{H}(X, Y; Z, W)$ then $\mathcal{H}(W, Z; Y, X)$;
29. If $\overrightarrow{XY}/\overrightarrow{XZ} = r$, Z is on p , and Y is not on p , then X is on *homothety* $_{Y, r/(1-r)}(p)$.

All listed lemmas are relatively simple and are often taught in primary or secondary schools within first lectures on “triangle geometry”. They can be proved using a Hilbert’s style geometry axioms or by algebraic theorem provers.

4.3 Primitive Constructions

We consider the following primitive construction steps:

1. Given distinct points X and Y it is possible to construct the line XY ;
2. Given distinct points X and Y it is possible to construct $k(X, Y)$;
3. Given two distinct lines/a line and a circle/two distinct circles that intersect it is possible to construct their common point(s);
4. Given distinct points X and Y it is possible to construct the side bisector of \overrightarrow{XY} ;
5. Given a point X and a line p it is possible to construct the line q that passes through X and is perpendicular to p ;
6. Given distinct points X and Y it is possible to construct the circle with diameter \overrightarrow{XY} ;
7. Given three distinct points it is possible to construct the circle that contains them all;

8. Given points X and Y and an angle α it is possible to construct the set of (all) points S such that $\angle XSY = \alpha$;
9. Given a point X and a line p it is possible to construct the point $s_p(X)$;
10. Given a line p and point X that does not lie on p it is possible to construct the circle k with center X that touches p ;
11. Given a point X outside a circle k it is possible to construct the line p that passes through X and touches k ;
12. Given two half-lines with the common initial point, it is possible to construct an angle congruent to the angle they constitute;
13. Given two intersecting lines it is possible to construct the bisector of internal angle they constitute;
14. Given one side of an angle and its internal angle bisector it is possible to construct the other side of the angle;
15. Given a point X , a line p and a rational number r , it is possible to construct the line *homothety* $_{X,r}(p)$;
16. Given points X , Y , and Z , and a rational number r it is possible to construct the point U such that $\overrightarrow{UX}/\overrightarrow{YZ} = r$.

All of the above construction steps can be (most of them trivially) expressed in terms of straightedge and compass operations. Still, for practical reasons, we use the above set instead of elementary straightedge and compass operations. These practical reasons are both more efficient search and simpler, high-level and more intuitive solutions.

5 Search Algorithm

Before the solving process, the preprocessing phase is performed on the above list of definitions and lemmas. For a fixed triangle ABC all points defined in Section 4.1 are uniquely determined (i.e., all definitions are instantiated). We distinguish between two types of lemmas:

instantiated Lemmas: lemmas that describe properties of one or a couple of fixed objects (lemmas 1-20).

generic Lemmas: lemmas given in an implication form (lemmas 21-29). These lemmas are given in a generic form and they are instantiated in a preprocessing phase by all objects satisfying their preconditions. So the instantiations are restricted with respect to the facts appearing in the definitions or lemmas.

Primitive constructions are given in a generic, non-instantiated form and they get instantiated while seeking for a construction sequence in the following manner: if there is an instantiation such that all objects from the premises of the primitive construction are already constructed (or given by a specification of the problem) then the instantiated object from the conclusion is constructed, if not already constructed. However, only a restricted set of objects is constructed – the objects appearing in some of the definitions or lemmas. For example, let us consider

the primitive construction stating that for two given points it is possible to construct the bisector of the segment they constitute. If there would be no restrictions, the segment bisector would be constructed for each two constructed points, while many of them would not be used anywhere further. In contrast, this rule would be applied only to a segment for which its bisector occurs in some of the definitions or lemmas (for instance, when the endpoints of the segment belong to a circle, so the segment bisector gives a line to which the center of the circle belongs to). This can reduce search time significantly, as well as a length of generated constructions.

The goal of the search procedure is to reach all points required by the input problem (for instance, for all Wernick's problems, the goal is the same: construct a triangle ABC , i.e., the points A , B and C). The search procedure is iterative – in each step it tries to apply a primitive construction to the known objects (given by the problem specification or already constructed) and if it succeeds, the search restarts from the first primitive construction, in the waterfall manner. If all required points are constructed, the search stops. If no primitive construction can be applied, the procedure stops with a failure. The efficiency of solving, and also the found solution may depend on the order in which the primitive constructions are listed.

We implemented the above procedure in Prolog⁴. At this point the program can solve 58 Wernick's problems, each in less than 1s (for other solvable problems it needs some additional lemmas)⁵. Of course, even with the above restricted search there are redundant construction steps performed during the search process and once the construction is completed, all these unnecessary steps are eliminated from the final, "clean" construction. The longest final construction consists of 11 primitive construction steps. Most of these "clean" constructions are the same as the ones that can be found in the literature. However, for problems with several different solutions, the one found by the system depends on the order of available primitive constructions/definitions/lemmas (one such example is given in Section 5.1). Along with the construction sequence, the set of non-degeneracy conditions (conditions that ensure that the constructed objects do exist, associated with some of construction primitives) is maintained.

5.1 Output

Once a required construction is found and simplified, it can be exported to different formats. Currently, export to simple natural language form is supported. For example, the construction for problem 7 (A, B, H) is represented as follows:

1. Using the point A and the point H , construct the line AH_a ;
2. Using the point B and the point H , construct the line BH_b ;
3. Using the point A and the line BH_b , construct the line AC ;
4. Using the point B and the line AH_a , construct the line BC ;
5. Using the line AC and the line BC , construct the point C .

⁴ The source code is available at <http://argo.matf.bg.ac.rs/?content=downloads>.

⁵ Currently, the program cannot solve the following solvable problems: 27, 43, 55, 57, 58, 69, 76, 82, 87, 96, 101, 126, 131, 139.

The generated construction can be (this is subject of our current work) also represented and illustrated (Figure 4) using the geometry language GCLC [21].

```
% free points
point A 5 5
point B 45 5
point H 32 18
% synthesized construction
line h_a A H
line h_b B H
perp b A h_b
perp a B h_a
intersec C a b
% drawing the triangle ABC
cmark_b A
cmark_b B
cmark_r C
cmark_b H
drawsegment A B
drawsegment A C
drawsegment B C
drawdashline h_a
drawdashline h_b
```

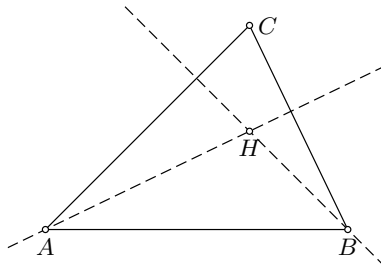


Fig. 4. A GCLC representation (left) and the corresponding illustration (right) for the solution to Wernick's problem 7

The above automatically generated solution is also example of a different (and simpler) solution from the one that can be found on the Internet [30] and that we used in building of our system (slightly reformulated in the way to use our set of primitive constructions):

1. Using the point A and the point B , construct the line AB ;
2. Using the point H and the line AB , construct the line CH_c ;
3. Using the line AB and the line CH_c , construct the point H_c ;
4. Using the point H and the line AB , construct the point H'_{AB} ;
5. Using the point A , the point B and point H'_{AB} , construct the circle $k(O, A)$;
6. Using the circle $k(O, A)$ and the line CH_c , construct the point C .

5.2 Proving Constructions Correct

Generated constructions can be proved correct using provers available within the GCLC tool (the tool provide support for three methods for automated theorem proving in geometry: the area method, Wu's method, and the Gröbner bases method) [19, 20]. For instance, the construction given in Section 5.1 in GCLC terms, can be verified using the following additional GCLC code (note that the given coordinates of the points A , B and H are used only for generating an illustration and are not used in the proving process):

```

% definition of the orthocenter
line _a B C
perp _h_a A _a
line _b A C
perp _h_b B _b
intersec _H _h_a _h_b
% verification
prove { identical H _H }
% alternatively
% prove { perpendicular A H B C }
% prove { perpendicular B H A C }

```

The conjecture that H is indeed the orthocenter of ABC was proved by Wu's method in 0.01s. Instead of proving that H is identical to the orthocenter, one could prove that it meets all conditions from the definition of the orthocenter (which can be more suitable, in terms of efficiency, for automated theorem provers). For example, the area method proves such two conditions in 0.04s. It also returns non-degeneracy conditions [22] (needed in the discussion phase): A , B and H are not collinear, neither of the angles BAH , ABH is right angle (additional conditions, such as the condition that the lines a and b from the GCLC construction intersect, are consequences of these conditions). If A and B are distinct, and A and H are identical, then any point C on the line passing through A and perpendicular to AB makes a solution, and similar holds if B and H are identical. Otherwise, if A , B , and H are pairwise distinct and collinear or one of the angles BAH and ABH is right angle, there are no solutions.

5.3 Re-evaluation

The presented approach focuses on one sort of triangle construction problems, but it can be used for other sorts of problems. We re-evaluated our approach on another corpus of triangle construction problems (discussed in Section 3). In each problem from this corpus, a task is to construct a triangle given three lengths from the following set of 9 lengths of characteristic segments in the triangle:

1. $|AB|$, $|BC|$, $|AC|$: lengths of the sides;
2. $|AM_a|$, $|BM_b|$, $|CM_c|$: lengths of the medians;
3. $|AH_a|$, $|BH_b|$, $|CH_c|$: lengths of the altitudes.

There are 20 significantly different problems in this corpus and they are all solvable. This family of problems is substantially different from Wernick's problems: in Wernick's problems, the task is to construct a triangle based on the given, located points, while in these problems, the task is to construct a triangle with some quantities equal to the given ones (hence, the two solutions to the problem are considered identical if the obtained triangles are congruent). However, it turns out that the underlying geometry knowledge is mostly shared [4, 5, 11, 12, 24]. We succeeded to solve 17 problems from this family, using the system described above and additional 9 defined points, 2 lemmas, and 8 primitive constructions. Extensions of the above list of primitive constructions was

expected because of introduction of segment measures. Since a search space was expanded by adding this new portion of knowledge, search times increased (the average solving time was 10s) and non-simplified constructions were typically longer than for the first corpus. However, simplified constructions are comparable in size with the ones from the first corpus and also readable.

6 Future Work

We plan to work on other corpora of triangle construction problems as well.⁶ In order to control the search space, the solving system should first detect the family to which the problem belongs and use only the corresponding rules. Apart from detecting needed high-level lemmas and rules, we will try to more deeply explore these lemmas and rules and derive them from (suitable) axioms and from elementary straightedge and compass construction steps.

The presented system synthesizes a construction and can use an external automated theorem prover to prove that the construction meets the specification (as described in Section 5; full automation of linking the solver with automated theorem provers is subject of our current work). However, the provers prove only statements of the form: “if the conditions Γ are met, then the specification is met as well”. They cannot, in a general case, check if the construction, the conditions Γ , are consistent (i.e., if the points that are constructed do exist). For instance, some provers cannot check if an intersection of two circles always exist. We are planning to use proof assistants and our automated theorem prover for coherent logic [32] for proving that the constructed points indeed exist (under generated non-degenerate conditions). With the verified theorem prover based on the area method [22] or with (more efficient) algebraic theorem proving verified by certificates [25], this would lead to completely machine verifiable solutions of triangle construction problems.

7 Conclusions

In our work we set up rather concrete tasks: (i) detect geometry knowledge needed for solving one of the most studied problems in mathematical education — triangle construction problems; (ii) develop a practical system for solving most of these problems. To our knowledge, this is the first systematic approach to deal with one family of problems (more focused than general construction problems) and to systemize underlying geometric knowledge. Our current results lead to a relatively small set of needed definitions, lemmas, and suitable primitive constructions and to a simple solving procedure. Generated constructions can be verified using external automated theorem provers. We believe that

⁶ In our preliminary experiments, our system solved all triangle construction problems (5 out of 25) in the corpus considered by Gulwani et.al [16]; our system can currently solve only a fraction of 135 problems considered by Gao and Chou [13], since most of them are not triangle construction problems or involve the knowledge still not supported by our system.

any practical solver would need to treat this detected geometry knowledge one way or another (but trading off with efficiency). We expect that limited additions to the the geometry knowledge presented here would enable solving most of triangle construction problems appearing in the literature.

Acknowledgments. We are grateful to prof. Pascal Schreck and to prof. Xiao-Shan Gao for providing us lists of construction problems solved by their systems and for useful feedback.

References

- [1] Adler, A.: *Theorie der geometrischen konstruktionen*, Göschen (1906)
- [2] Anglesio, J., Schindler, V.: Solution to problem 10719. *American Mathematical Monthly* 107, 952–954 (2000)
- [3] Beeson, M.: Constructive geometry. In: *Proceedings of the Tenth Asian Logic Colloquium*. World Scientific (2010)
- [4] Berzsenyi, G.: Constructing triangles from three given parts. *Quantum* 396 (July/August 1994)
- [5] Connelly, H.: An extension of triangle constructions from located points. *Forum Geometricorum* 9, 109–112 (2009)
- [6] Connelly, H., Dergiades, N., Ehrmann, J.-P.: Construction of triangle from a vertex and the feet of two angle bisectors. *Forum Geometricorum* 7, 103–106 (2007)
- [7] Danneels, E.: A simple construction of a triangle from its centroid, incenter, and a vertex. *Forum Geometricorum* 5, 53–56 (2005)
- [8] Davis, P.J.: The rise, fall, and possible transfiguration of triangle geometry: A mini-history. *The American Mathematical Monthly* 102(3), 204–214 (1995)
- [9] DeTemple, D.W.: Carlyle circles and the lemoine simplicity of polygon constructions. *The American Mathematical Monthly* 98(2), 97–108 (1991)
- [10] Djorić, M., Janičić, P.: Constructions, instructions, interactions. *Teaching Mathematics and its Applications* 23(2), 69–88 (2004)
- [11] Fursenko, V.B.: Lexicographic account of triangle construction problems (part i). *Mathematics in Schools* 5, 4–30 (1937)
- [12] Fursenko, V.B.: Lexicographic account of triangle construction problems (part ii). *Mathematics in schools* 6, 21–45 (1937)
- [13] Gao, X.-S., Chou, S.-C.: Solving geometric constraint systems. I. A global propagation approach. *Computer-Aided Design* 30(1), 47–54 (1998)
- [14] Gao, X.-S., Chou, S.-C.: Solving geometric constraint systems. II. A symbolic approach and decision of Rc-constructibility. *Computer-Aided Design* 30(2), 115–122 (1998)
- [15] Grima, M., Pace, G.J.: An Embedded Geometrical Language in Haskell: Construction, Visualisation, Proof. In: *Proceedings of Computer Science Annual Workshop* (2007)
- [16] Gulwani, S., Korthikanti, V.A., Tiwari, A.: Synthesizing geometry constructions. In: *Programming Language Design and Implementation, PLDI 2011*, pp. 50–61. ACM (2011)
- [17] Chen, G.: *Les Constructions Géométriques á la Règle et au Compas par une Méthode Algébrique*. Master thesis, University of Strasbourg (1992)

- [18] Holland, G.: Computerunterstützung beim Lösen geometrischer Konstruktionsaufgaben. ZDM Zentralblatt für Didaktik der Mathematik 24(4) (1992)
- [19] Janičić, P.: GCLC — A Tool for Constructive Euclidean Geometry and More Than That. In: Iglesias, A., Takayama, N. (eds.) ICMS 2006. LNCS, vol. 4151, pp. 58–73. Springer, Heidelberg (2006)
- [20] Janičić, P., Quaresma, P.: System Description: GCLCprover + GeoThms. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 145–150. Springer, Heidelberg (2006)
- [21] Janičić, P.: Geometry Constructions Language. Journal of Automated Reasoning 44(1-2), 3–24 (2010)
- [22] Janičić, P., Narboux, J., Quaresma, P.: The area method: a recapitulation. Journal of Automated Reasoning 48(4), 489–532 (2012)
- [23] Lebesgue, H.-L.: Leçons sur les constructions géométriques. Gauthier-Villars (1950)
- [24] Lopes, L.: Manuel de Construction de Triangles. QED Texte (1996)
- [25] Marić, F., Petrović, I., Petrović, D., Janičić, P.: Formalization and implementation of algebraic methods in geometry. Electronic Proceedings in Theoretical Computer Science 79 (2012)
- [26] Martin, G.E.: Geometric Constructions. Springer (1998)
- [27] Meyers, L.F.: Update on William Wernick’s “triangle constructions with three located points”. Mathematics Magazine 69(1), 46–49 (1996)
- [28] Pambuccian, V.: Axiomatizing geometric constructions. Journal of Applied Logic 6(1), 24–46 (2008)
- [29] Schreck, P.: Constructions à la règle et au compas. PhD thesis, University of Strasbourg (1993)
- [30] Specht, E.: Wernicks liste (in German), <http://hydra.nat.uni-magdeburg.de/wernick/>
- [31] Stewart, I.: Galois Theory. Chapman and Hall Ltd. (1973)
- [32] Stojanović, S., Pavlović, V., Janičić, P.: A Coherent Logic Based Geometry Theorem Prover Capable of Producing Formal and Readable Proofs. In: Schreck, P., Narboux, J., Richter-Gebert, J. (eds.) ADG 2010. LNCS, vol. 6877, pp. 201–220. Springer, Heidelberg (2011)
- [33] Ustinov, A.V.: On the construction of a triangle from the feet of its angle bisectors. Forum Geometricorum 9, 279–280 (2009)
- [34] Wernick, W.: Triangle constructions with three located points. Mathematics Magazine 55(4), 227–230 (1982)
- [35] Yiu, P.: Elegant geometric constructions. Forum Geometricorum 5, 75–96 (2005)

A Query Language for Formal Mathematical Libraries

Florian Rabe

Jacobs University Bremen, Germany

Abstract. One of the most promising applications of mathematical knowledge management is search: Even if we restrict attention to the tiny fragment of mathematics that has been formalized, the amount exceeds the comprehension of an individual human.

Based on the generic representation language MMT, we introduce the mathematical query language QMT: It combines simplicity, expressivity, and scalability while avoiding a commitment to a particular logical formalism. QMT can integrate various search paradigms such as unification, semantic web, or XQuery style queries, and QMT queries can span different mathematical libraries.

We have implemented QMT as a part of the MMT API. This combination provides a scalable indexing and query engine that can be readily applied to any library of mathematical knowledge. While our focus here is on libraries that are available in a content markup language, QMT naturally extends to presentation and narration markup languages.

1 Introduction and Related Work

Mathematical knowledge management applications are particularly strong at large scales, where automation can be significantly superior to human intuition. This makes search and retrieval pivotal MKM applications: The more the amount of mathematical knowledge grows, the harder it becomes for users to find relevant information. Indeed, even expert users of individual libraries can have difficulties reusing an existing development because they are not aware of it. Therefore, this question has received much attention.

Object query languages augment standard text search with phrase queries that match mathematical operators and with wild cards that match arbitrary mathematical expressions. Abstractly, an object query engine is based on an index, which is a set of pairs (l, o) where o is an object and l is the location of o . The index is built from a collection of mathematical documents, and the result of an object query is a subset of the index. The object model is usually based on presentation MathML and/or content MathML/OpenMath [W3C03,BCC+04], but importers can be used to index other formats such as LaTeX. Examples for object query languages and engines are given in [MY03,MM06,MG08,KS06,SL11]. A partial overview can be found in [SL11]. A central question is the use of wild cards. An example language with complex wild cards is given in [AY08]. Most

generally, [KS06] uses unification queries that return all objects that can be unified with the query.

Property query languages are similar to object query languages except that both the index and the query use relational information that abstracts from the mathematical objects. For example, the relational index might store the toplevel symbol of every object or the “used-in” relation between statements. This approximates an object index, and many property queries are special cases of object queries. But property queries are simpler and more efficient, and they still cover many important examples. Such languages are given in [GC03, AS04] and [BR03] based on the Coq and Mizar libraries, respectively.

Compositional query languages focus on a complex language of query expressions that are evaluated compositionally. The atomic queries are provided by the elements of the queried library. SQL [ANS03] uses n -ary relations between elements, and query expressions use the algebra of relations. The SPARQL [W3C08] data model is RDF, and queries focus on unary and binary predicates on a set of URIs of statements. This could serve as the basis for mathematics on the semantic web. Both data models match bibliographical meta-data and property-based indices and could also be applied to the results of object queries (seen as sets of pairs); but they are not well-suited for expressions. The XQuery [W3C07] data model is XML, and query expressions are centered around operations on lists of XML nodes. This is well-suited for XML-based markup languages for mathematical documents and expressions and was applied to OMDOC [Koh06] in [ZK09]. In [KRZ10], the latter was combined with property queries. Very recently [ADL12] gave a compositional query language for hiproof proof trees that integrates concepts from both object and property queries.

A number of **individual libraries** of mathematics provide custom query functionality. Object query languages are used, for example, in [LM06] for Activemath or in Wolfram|Alpha. Most interactive proof assistants permit some object or property queries, primarily to search for theorems that are applicable to a certain goal, e.g., Isabelle, Coq, and Matita. [Urb06] is notable for using automated reasoning to prepare an index of all Mizar theorems.

It is often desirable to combine several of the above formalisms in the same query. Therefore, we have designed the query language QMT with the goal of permitting as many different query paradigms as possible. QMT uses a simple kernel syntax in which many advanced query paradigms can be defined. This permits giving a formal syntax, a formal semantics, and a scalable implementation, all of which are presented in this paper.

QMT is grounded in the MMT language (Module System for Mathematical Theories) [RK11], a scalable, modular representation language for mathematical knowledge. It is designed as a scalable trade-off between (i) a logical framework with formal syntax and semantics and (ii) an MKM framework that does not commit to any particular formal system. Thus, MMT permits both adequate representations of virtually any formal system as well as the implementation of generic MKM services. We implement QMT on top of our MMT system, which provides a flexible and scalable query API and query server.

Declaration	Intended Semantics
base type a	a set of objects
concept symbol c	a subset of a base type
relation symbol r	a relation between two base types
function symbol f	a sorted first-order function
predicate symbol p	a sorted first-order predicate

Kind of Expression	Intended Semantics
Type T	a set
Query $Q : T$	an element of T
element query $Q : t$	an element of t
set query $Q : \text{set}(t)$	a subset of t
Relation $R < a, a'$	a relation between a and a'
Proposition F	a boolean truth value

Fig. 1. QMT Notions and their Intuitions

Our design has two pivotal strengths. Firstly, QMT can be applied to the libraries of any formal system that is represented as MMT. Queries can even span libraries of different systems. Secondly, QMT queries can make use of other MMT services. For example, queries can access the inferred type and the presentation of a found expression, which are computed dynamically.

We split the definition of QMT into two parts. Firstly, Sect. 2 defines QMT signatures in general and then the syntax and semantics of QMT for an arbitrary signature. Secondly, Sect. 3 describes a specific QMT signature that we use for MMT libraries. Our implementation, which is based on that signature, is presented in Sect. 4.

2 The QMT Query Language

2.1 Syntax

Our syntax arises by combining features of sorted first-order logic – which leads to very intuitive expressions – and of description logics – which leads to efficient evaluations. Therefore, our **signatures** Σ contain five kinds of declarations as given in Fig. 1.

For a given signature, we define four kinds of **expressions**: types T , relations R , propositions F , and typed queries Q as listed in Fig. 1. The grammar for signatures and expressions is given in Fig. 2.

The intuitions for most expression formation operators can be guessed easily from their notations. In the following we will discuss each in more detail.

Regarding **types** T , we use product types and power type. However, we go out of our way to avoid arbitrary nestings of type constructors. Every type is either a product $t = a_1 \times \dots \times a_n$ of base types a_i or the power type $\text{set}(t)$ of such a type. Thus, we are able to use the two most important type formation operators in the context of querying: product types arise when a query contains multiple query variables, and power types arise when a query returns multiple

Signatures	$\Sigma ::= \cdot \mid \Sigma, a : \text{type} \mid \Sigma, c < a \mid \Sigma, r < a, a$ $\mid \Sigma, f : T, \dots, T \rightarrow T \mid \Sigma, p : T, \dots, T \rightarrow \text{prop}$
Contexts	$\Gamma ::= \cdot \mid \Gamma, x : T$
Simple Types	$t ::= a \times \dots \times a$
General Types	$T ::= t \mid \text{set}(t)$
Relations	$R ::= r \mid R^{-1} \mid R^* \mid R; R; R \mid R \cup R \mid R \cap R \mid R \setminus R$
Propositions	$F ::= p(Q, \dots, Q) \mid \neg F \mid F \wedge F \mid \forall x \in Q. F(x)$
Queries	$Q ::= c \mid x \mid f(Q, \dots, Q) \mid Q * \dots * Q \mid Q_i$ $\mid R(Q) \mid \bigcup_{x \in Q} Q(x) \mid \{x \in Q \mid F(x)\}$

Fig. 2. The Grammar for Query Expressions

results. But at the same time, the type system remains very simple and can be treated as essentially first-order.

Regarding **relations**, we provide the common operations from the calculus of binary relations: dual/inverse R^{-1} , transitive closure R^* , composition $R; R'$, union $R \cup R'$, intersection $R \cap R'$, and difference $R \setminus R'$. Notably absent is the absolute complement operation R^c ; it is omitted because its semantics can not be computed efficiently in general. Note that the operation R^{-1} is only necessary for atomic R : For all other cases, we can put $(R^*)^{-1} = (R^{-1})^*$, $(R; R')^{-1} = R'^{-1}; R^{-1}$, and $(R * R')^{-1} = R^{-1} * R'^{-1}$ for $*$ $\in \{\cup, \cap, \setminus\}$.

Regarding **propositions**, we use the usual constructors of classical first-order logic: predicates, negation, conjunction, and universal quantification. As usual, the other constructors are definable. However, there is one specialty: The quantification $\forall x \in Q. F(x)$ does not quantify over a type t ; instead, it is relativized by a query result $Q : \text{set}(t)$. This specialty is meant to support efficient evaluation: The extension of a base type is usually much larger than that of a query, and it may not be efficiently computable or not even finite.

Regarding **queries**, our language combines intuitions from description and first-order logic with an intuitive mathematical notation. Constants c , variables x , and function application are as usual for sorted first-order logic. $Q^1 * \dots * Q^n$ for $n \in \mathbb{N}$ and Q_i for $i = 1, \dots, n$ denote tupling and projection. $R(Q)$ represents the image of the object given by Q under the relation given by R . $\bigcup_{x \in Q} Q'(x)$ denotes the union of the family of queries $Q'(x)$ where x runs over all objects in the result of Q . Finally, $\{x \in Q \mid F(x)\}$ denotes comprehension on queries, i.e., the objects in Q that satisfy F . Just like for the universal quantification, all bound variables are relativized to a query result to support efficient evaluation.

Remark 1. While we do not present a systematic analysis of the efficiency of QMT, we point out that we designed the syntax of QMT with the goal of supporting efficient evaluation. In particular, this motivated our distinction between the ontology part, i.e., concept and relation symbols, and the first-order part, i.e., the function and predicate symbols.

Indeed, every concept $c < t$ can be regarded as a function symbol $c : \text{set}(t)$, and every relation $r < a, a'$ as a predicate symbol $r : a, a' \rightarrow \text{prop}$. Thus, the ontology symbols may appear redundant — their purpose is to permit efficient evaluations. This is most apparent for relations. For a predicate symbol $p :$

$\frac{n \text{ not declared in } \Sigma}{n \notin \Sigma}$	$\frac{}{\vdash \cdot}$	$\frac{\vdash \Sigma \quad a \notin \Sigma}{\vdash \Sigma, a : \text{type}}$
$\frac{\vdash \Sigma \quad c \notin \Sigma \quad a : \text{type in } \Sigma}{\vdash \Sigma, c < a}$	$\frac{\vdash \Sigma \quad r \notin \Sigma \quad (a_i : \text{type in } \Sigma)_{i=1}^2}{\vdash \Sigma, r < a_1, a_2}$	
$\frac{\vdash \Sigma \quad f \notin \Sigma \quad (\vdash_{\Sigma} T_i : \text{type})_{i=1}^{n+1}}{\vdash \Sigma, f : T_1, \dots, T_n \rightarrow T_{n+1}}$	$\frac{\vdash \Sigma \quad p \notin \Sigma \quad (\vdash_{\Sigma} T_i : \text{type})_{i=1}^n}{\vdash \Sigma, p : T_1, \dots, T_n \rightarrow \text{prop}}$	

Fig. 3. Well-Formed Signatures

$a, a' \rightarrow \text{prop}$, evaluation requires a method that maps from $\llbracket a \rrbracket \times \llbracket a' \rrbracket$ to booleans. But for a relation symbol $r < a, a'$, evaluation requires a method that returns for any u all v such that $(u, v) \in \llbracket r \rrbracket$ or all v such that $(v, u) \in \llbracket r \rrbracket$. A corresponding property applies to concepts.

Therefore, efficient implementations of QMT should maintain indices for them that are computed a priori: hash sets for the concept symbols and hash tables for the relation symbols. (Note that using hash tables for all relation symbols permits fast evaluation of all relation expressions R , which is crucial for the evaluation of queries $R(Q)$.) The implementation of function and predicate symbols, on the other hand, only requires plain functions that are called when evaluating a query.

Thus, it is a design decision whether a certain feature is realized by an ontology or by a first-order symbol. By separating the ontology and the first-order part, we permit simple indices for the former and retain flexible extensibility for the latter (see also Rem. 2).

Based on these intuitions, it is straightforward to define the **well-formed expressions**, i.e., the expressions that will have a denotational semantics. More formally, we use the **judgments** given in Fig. 5

Judgment	Intuition
$\vdash \Sigma$	well-formed signature Σ
$\vdash_{\Sigma} T : \text{type}$	well-formed type T
$\Gamma \vdash_{\Sigma} Q : T$	well-typed query Q of type T
$\Gamma \vdash_{\Sigma} Q : T$	well-typed query Q of type T
$\vdash_{\Sigma} R < a, a'$	well-typed relation R between a and a'
$\Gamma \vdash_{\Sigma} F : \text{prop}$	well-formed proposition F

Fig. 5. Judgments

to define the well-formed expressions over a signature Σ and a context Γ . The **rules** for these judgments are given in Fig. 3 and 4.

In order to give some meaningful examples, we will already make use of the symbols from the MMT signature, which we will introduce in Sect. 3.

$\frac{(a_i : \text{type in } \Sigma)_{i=1}^n}{\vdash_{\Sigma} a_1 \times \dots \times a_n : \text{type}}$	$\frac{(a_i : \text{type in } \Sigma)_{i=1}^n}{\vdash_{\Sigma} \text{set}(a_1 \times \dots \times a_n) : \text{type}}$	
$\frac{c < t \text{ in } \Sigma}{\Gamma \vdash_{\Sigma} c : \text{set}(t)}$	$\frac{f : T_1, \dots, T_n \rightarrow T \text{ in } \Sigma \quad \Gamma \vdash_{\Sigma} Q_i : T_i}{\Gamma \vdash_{\Sigma} f(Q_1, \dots, Q_n) : T}$	$\frac{x : T \text{ in } \Gamma}{\Gamma \vdash_{\Sigma} x : T}$
$\frac{\Gamma \vdash_{\Sigma} Q_i : t_i \text{ for } i = 1, \dots, n}{\Gamma \vdash_{\Sigma} Q_1 * \dots * Q_n : t_1 \times \dots \times t_n}$	$\frac{\Gamma \vdash_{\Sigma} Q : t_1 \times \dots \times t_n \quad i \in \{1, \dots, n\}}{\Gamma \vdash_{\Sigma} Q_i : t_i}$	
$\frac{\Gamma \vdash_{\Sigma} Q : \text{set}(t) \quad \Gamma, x : t \vdash_{\Sigma} Q'(x) : \text{set}(t')}{\Gamma \vdash_{\Sigma} \bigcup_{x \in Q} Q'(x) : \text{set}(t')}$		
$\frac{\Gamma \vdash_{\Sigma} Q : t \quad \vdash_{\Sigma} R < t, t'}{\Gamma \vdash_{\Sigma} R(Q) : \text{set}(t')}$	$\frac{\Gamma \vdash_{\Sigma} Q : \text{set}(t) \quad \Gamma, x : t \vdash_{\Sigma} F(x) : \text{prop}}{\Gamma \vdash_{\Sigma} \{x \in Q \mid F(x)\} : \text{set}(t)}$	
$\frac{r < a, a' \text{ in } \Sigma}{\vdash_{\Sigma} r < a, a'}$	$\frac{\vdash_{\Sigma} R < a, a'}{\vdash_{\Sigma} R^{-1} < a', a}$	$\frac{\vdash_{\Sigma} R < a, a}{\vdash_{\Sigma} R^* < a, a}$
$\frac{\vdash_{\Sigma} R < a, a' \quad \vdash_{\Sigma} R' < a', a''}{\vdash_{\Sigma} R; R' < a, a''}$	$\frac{\vdash_{\Sigma} R < a, a' \quad \vdash_{\Sigma} R' < a, a' \quad * \in \{\cup, \cap, \setminus\}}{\vdash_{\Sigma} R * R' < a, a'}$	
$\frac{p : T_1, \dots, T_n \rightarrow \text{prop} \text{ in } \Sigma \quad \Gamma \vdash_{\Sigma} Q_i : T_i}{\Gamma \vdash_{\Sigma} p(Q_1, \dots, Q_n) : \text{prop}}$		
$\frac{\Gamma \vdash_{\Sigma} F : \text{prop}}{\Gamma \vdash_{\Sigma} \neg F : \text{prop}}$	$\frac{\Gamma \vdash_{\Sigma} F : \text{prop} \quad \Gamma \vdash_{\Sigma} F' : \text{prop}}{\Gamma \vdash_{\Sigma} F \wedge F' : \text{prop}}$	
$\frac{\Gamma \vdash_{\Sigma} Q : \text{set}(t) \quad \Gamma, x : t \vdash_{\Sigma} F(x) : \text{prop}}{\Gamma \vdash_{\Sigma} \forall x \in Q. F(x) : \text{prop}}$		

Fig. 4. Well-Formed Expressions

Example 1. Consider a base type $id : \text{type}$ of MMT identifiers in some fixed MMT library. Moreover, consider a concept symbol $\text{theory} < id$ giving the identifiers of all theories, and a relation symbol $\text{includes} < id, id$ that gives the relation “theory A directly includes theory B ”.

Then the query *theory* of type $set(id)$ yields the set of all theories. Given a theory u , the query $includes^{*-1}(u)$ of type $set(id)$ yields the set of all theories that transitively include u .

Example 2 (Continued). Additionally, consider a concept $constant < id$ of identifiers of MMT constants, relation symbol $declares < id, id$ that relates every theory to the constants declared in it, a base type $obj : type$ of OPENMATH objects, a function symbol $type : id \rightarrow obj$ that maps each MMT constant to its type, and a predicate symbol $occurs : id, obj \rightarrow prop$ that determines whether an identifier occurs in an object.

Then the following query of type $set(id)$ retrieves all constants that are included into the theory u and whose type uses the identifier v :

$$\{x \in (includes^*; declares)(u) \mid occurs(v, type(x))\}$$

2.2 Semantics

A Σ -model assigns to every symbol s in Σ a denotation. The formal definition is given in Def. 1. Relative to a fixed model M (which we suppress in the notation), each well-formed expression has a well-defined denotational semantics, given by the interpretation function $\llbracket - \rrbracket$. The semantics of propositions and queries in context Γ is relative to an assignment α , which assigns values to all variables in Γ . An overview is given in Fig. 6. The formal definition is given in Def. 2.

Judgment	Semantics
$\vdash_{\Sigma} T : type$	$\llbracket T \rrbracket \in \mathcal{SET}$
$\Gamma \vdash_{\Sigma} Q : t$	$\llbracket Q \rrbracket^{\alpha} \in \llbracket t \rrbracket$
$\Gamma \vdash_{\Sigma} Q : set(t)$	$\llbracket Q \rrbracket^{\alpha} \subseteq \llbracket t \rrbracket$
$\vdash_{\Sigma} R < a, a'$	$\llbracket R \rrbracket \subseteq \llbracket a \rrbracket \times \llbracket a' \rrbracket$
$\Gamma \vdash_{\Sigma} F : prop$	$\llbracket F \rrbracket^{\alpha} \in \{0, 1\}$

Fig. 6. Semantics of Judgments

Definition 1 (Models). A Σ -model M assigns to every Σ -symbol s a denotation s^M such that

- a^M is a set for $a : type$
- $c^M \subseteq \llbracket a \rrbracket$ for $c < a$
- $r^M \subseteq \llbracket a \rrbracket \times \llbracket a' \rrbracket$ for $r < a, a'$
- $f^M : \llbracket T_1 \rrbracket \times \dots \times \llbracket T_n \rrbracket \rightarrow \llbracket T \rrbracket$ for $f : T_1, \dots, T_n \rightarrow T$
- $p^M : \llbracket T_1 \rrbracket \times \dots \times \llbracket T_n \rrbracket \rightarrow \{0, 1\}$ for $p : T_1, \dots, T_n \rightarrow prop$

Definition 2 (Semantics). Given a Σ -model M , the interpretation function $\llbracket - \rrbracket$ is defined as follows.

Semantics of types:

- $\llbracket a_1 \times \dots \times a_n \rrbracket$ is the cartesian product $a_1^M \times \dots \times a_n^M$
- $\llbracket set(t) \rrbracket$ is the power set of $\llbracket t \rrbracket$

Semantics of relations:

- $\llbracket r \rrbracket = r^M$
- $\llbracket R^{-1} \rrbracket$ is the dual/inverse relation of $\llbracket R \rrbracket$, i.e., the set $\{(u, v) \mid (v, u) \in \llbracket R \rrbracket\}$

- R^* is the transitive closure of $\llbracket R \rrbracket$
- $R; R'$ is the composition of $\llbracket R \rrbracket$ and $\llbracket R' \rrbracket$,
i.e., the set $\{(u, w) \mid \exists v \text{ such that } (u, v) \in \llbracket R \rrbracket, (v, w) \in \llbracket R' \rrbracket\}$
- $R \cup R'$, $R \cap R'$, and $R \setminus R'$ are interpreted in the obvious way using the union, intersection, and difference of sets

Semantics of propositions under an assignment α :

- $\llbracket p(Q_1, \dots, Q_n) \rrbracket^\alpha = p^M(\llbracket Q_1 \rrbracket^\alpha, \dots, \llbracket Q_n \rrbracket^\alpha)$
- $\llbracket \neg F \rrbracket^\alpha = 1$ iff $\llbracket F \rrbracket^\alpha = 0$
- $\llbracket F \wedge F' \rrbracket^\alpha = 1$ iff $\llbracket F \rrbracket^\alpha = 1$ and $\llbracket F' \rrbracket^\alpha = 1$
- $\llbracket \forall x \in Q. F(x) \rrbracket^\alpha = 1$ iff $\llbracket F(x) \rrbracket^{\alpha, x/u} = 1$ for all $u \in \llbracket Q \rrbracket^\alpha$

Semantics of queries $\Gamma \vdash_\Sigma Q : T$ under an assignment α :

- $\llbracket c \rrbracket^\alpha = c^M$
- $\llbracket x \rrbracket^\alpha = \alpha(x)$
- $\llbracket f(Q_1, \dots, Q_n) \rrbracket^\alpha = f^M(\llbracket Q_1 \rrbracket^\alpha, \dots, \llbracket Q_n \rrbracket^\alpha)$
- $\llbracket R(Q) \rrbracket^\alpha = \{u \in \llbracket a' \rrbracket^\alpha \mid (\llbracket Q \rrbracket^\alpha, u) \in \llbracket R \rrbracket\}$ for a relation $\vdash_\Sigma R < a, a'$ and a query $\Gamma \vdash_\Sigma Q : a$
informally, $\llbracket R(Q) \rrbracket^\alpha$ is the image of $\llbracket Q \rrbracket^\alpha$ under $\llbracket R \rrbracket$
- $\llbracket \bigcup_{x \in Q} Q'(x) \rrbracket^\alpha$ is the union of all sets $\llbracket Q'(x) \rrbracket^{\alpha, x/u}$ where u runs over all elements of $\llbracket Q \rrbracket^\alpha$
- $\llbracket \{x \in Q \mid F(x)\} \rrbracket^\alpha$ is the subset of $\llbracket Q \rrbracket^\alpha$ containing all elements u for which $\llbracket F(x) \rrbracket^{\alpha, x/u} = 1$

Remark 2. It is easy to prove that if all concept and relation symbols are interpreted as finite sets and if all function symbols with result type $set(t)$ always return finite sets, then all well-formed queries of type $set(t)$ denote a *finite* subset of $\llbracket t \rrbracket$. Moreover, if the interpretations of the function and predicate symbols are computable functions, then the interpretation of queries is computable as well. This holds even if base types are interpreted as infinite sets.

2.3 Predefined Symbols

We use a number of predefined function and predicate symbols as given in Fig. 7. These are assumed to be implicitly declared in every signature, and their semantics is fixed. All of these symbols are overloaded for all simple types t . Moreover, we use special notations for them.

Symbol	Type	Semantics
$\{-\}$	$t \rightarrow set(t)$	the singleton set
$- \doteq -$	$t, t \rightarrow prop$	equality
$- \in -$	$t, set(t) \rightarrow prop$	elementhood

Fig. 7. Predefined Symbols

All of this is completely analogous to the usual treatment of equality as a predefined predicate symbol in first-order logic. The only difference is that our slightly richer type system calls for a few additional predefined symbols.

It is easy to add further predefined symbols, in particular equality of sets (which, however, may be inefficient to decide) and binary union of queries. We omit these here for simplicity.

2.4 Definable Queries

Using the predefined symbols, we can define a number of further useful query formation operators:

Example 3. Using the singleton symbol $\{-\}$, we can define for $\Gamma \vdash_{\Sigma} Q : \text{set}(t)$ and $\Gamma, x : t \vdash_{\Sigma} q(x) : t'$

$$\{q(x) : x \in Q\} := \bigcup_{x \in Q} \{q(x)\} \quad \text{of type } \text{set}(t').$$

It is easy to show that, semantically, this is the replacement operator, i.e., $\llbracket \{q(x) : x \in Q\} \rrbracket^{\alpha}$ is the set containing exactly the elements $\llbracket q(x) \rrbracket^{\alpha, x/u}$ for any $u \in \llbracket Q \rrbracket^{\alpha}$.

Example 4 (SQL-style Queries). For a query $\vdash_{\Sigma} Q : \text{set}(a_1 \times \dots \times a_N)$, natural numbers $n_1, \dots, n_k \in \{1, \dots, N\}$, and a proposition $x_1 : a_1, \dots, x_N : a_N \vdash_{\Sigma} F(x_1, \dots, x_N) : \text{prop}$, we write

$$\text{select } n_1, \dots, n_k \text{ from } Q \text{ where } F(1, \dots, N)$$

for the query

$$\{x_{n_1} * \dots * x_{n_k} : x \in \{y \in Q \mid F(y_1, \dots, y_N)\}\}$$

of type $\text{set}(a_{n_1} \times \dots \times a_{n_k})$.

Example 5 (XQuery-style Queries). For queries $\vdash_{\Sigma} Q : \text{set}(a)$ and $x : a \vdash_{\Sigma} q'(x) : a'$ and $x : a, y : a' \vdash_{\Sigma} Q''(x, y) : \text{set}(a'')$, and a proposition $x : a, y : a' \vdash_{\Sigma} F(x, y) : \text{prop}$, we write

$$\text{for } x \text{ in } Q \text{ let } y = q'(x) \text{ where } F(x, y) \text{ return } Q''(x, y)$$

for the query

$$\bigcup_{z \in P} Q''(z_1, z_2) \quad \text{with} \quad P := \{z \in \{x * q'(x) : x \in Q\} \mid F(z_1, z_2)\}$$

of type $\text{set}(a'')$.

Example 6 (DL-style Queries). For a relation $\vdash_{\Sigma} R < a, a'$, a concept $c < a$, and a query $\vdash_{\Sigma} Q : \text{set}(a')$, we write $\square^c R.Q$ for the query $\{x \in c \mid \forall y \in R(x). y \in Q\}$ of type $\text{set}(a)$.

Note that, contrary to the universal restriction $\square R.Q$ in description logic, we have to restrict the query to all x of concept c instead of querying for all x of type a . This makes sense in our setting because we assume that we can only iterate efficiently over concepts but not over (possibly infinite!) base types.

However, this is not a loss of generality: individual signatures may always couple a base type a with a concept is_a such that $\llbracket is_a \rrbracket = \llbracket a \rrbracket$.

Declaration	Intuition
Base types	
id : <i>type</i>	URIs of MMT declarations
obj : <i>type</i>	MMT (OPENMATH) OBJECTS
xml : <i>type</i>	XML elements
Concepts	
$theory$ < id	theories
$view$ < id	views
$constant$ < id	constants
$style$ < id	styles
Relations	
$includes$ < id, id	inclusion between theories
$declares$ < id, id	declarations in a theory
$domain$ < id, id	domain of structure/view
$codomain$ < id, id	codomain of structure/view
Functions	
$type$: $id \rightarrow obj$	type of a constant
def : $id \rightarrow obj$	definiens of a constant
$infer$: $id, obj \rightarrow obj$	type inference relative to a theory
arg_p : $obj \rightarrow obj$	argument at position p
$subobj$: $obj, id \rightarrow set(obj)$	subobjects with a certain head
$unify$: $obj \rightarrow set(id \times obj \times obj)$	all objects that unify with a given one
$render$: $id, id \rightarrow xml$	rendering of a declaration using a certain style
$render$: $obj, id \rightarrow xml$	rendering of an object using a certain style
u : id	literals for MMT URIs u
o : obj	literals for MMT objects o
Predicates	
$occurs$: $id, obj \rightarrow prop$	occurs in

Fig. 8. The QMT Signature for MMT

3 Querying MMT Libraries

We will now fix an MMT-specific signature Σ that customizes QMT with the MMT ontology as well as with several functions and predicates based on the MMT specification. The declarations of Σ are listed in Fig. 8.

For simplicity, we avoid presenting any details of MMT and refer to [RK11] for a comprehensive description. For our purposes, it is sufficient to know that MMT organizes mathematical knowledge in a simple ontology that can be seen as the fragment of OMDOC pertaining to formal theories. We will explain the necessary details below when explaining the respective Σ -symbols.

An MMT **library** is any set of MMT declarations (not necessarily well-typed or closed under dependency). We will assume a fixed library L in the following. Based on L , we will define a model M by giving the interpretation s^M for every symbol s listed in Fig. 8.

Base Types. We use three base types. Firstly, every MMT declaration has a globally unique **canonical identifier**, its MMT URI. We use this to define id^M as the set of all MMT URIs declared in L .

obj^M the set of all OPENMATH objects that can be formed from the symbols in id^M . In order to handle objects with free variables conveniently, we use the following convention: All objects in obj^M are technically closed; but we permit the use of a special binder symbol **free**, which can be used to formally bind the free variables. This has the advantage that the context of an object, which may carry, e.g., type attributions, is made explicit. Using general OPENMATH objects means that the type obj is subject to exactly α -equality and attribution flattening, the only equalities defined in the OPENMATH standard. The much more difficult problem of queries relative to a stronger equality relation remains future work.

The remaining base type xml is a generic container for any non-MMT **XML data** such as HTML or presentation MathML. Thus, xml^M is the set of all XML elements. This is useful because the MMT API contains several functions that return XML.

Ontology. For simplicity, we restrict attention to the most important notions of the MMT ontology; adding the remaining notions is straightforward. The ontology only covers the MMT declarations, all of which have canonical identifiers. Thus, all concepts refine the type id , and all relations are between identifiers.

Among the MMT **concepts, theories** are used to represent logics, theories of a logic, ontologies, type theories, etc. They contain **constants**, which represent function symbols, type operators, inference rules, theorems, etc. Constants may have OPENMATH **objects** [BCC⁺04] as their type or definiens. Theories are related via theory morphisms called **views**. These are truth-preserving translations from one theory to another and represent translations and models. Theories and views together form a multi-graph of theories across which theorems can be shared. Finally, **styles** contain notations that govern the translation from content to presentation markup.

MMT theories, views, and styles can be structured by a strong module system. The most important modular construct is the *includes* relation for explicit imports. The *declares* relation relates every theory to the constants it declares; this includes the constants that are not explicitly declared in L but induced by the module system. Finally, two further relations connect each view to its *domain* and *codomain*.

All concepts and relations are interpreted in the obvious way. For example, the set $theory^M$ contains the MMT URIs of all theories in L .

Function and Predicate Symbols. Regarding the function and predicate symbols, we are very flexible because a wide range of operations can be defined for MMT libraries. In particular, every function implemented in the MMT API can be easily exposed as a Σ -symbol. Therefore, we only show a selection of symbols that showcase the potential.

In Sect. 2, we have deliberately omitted **partial function symbols** in order to simplify the presentation of our language. However, in practice, it is often necessary to add them. For example, def^M must be a partial function because (i) the argument might not be the MMT URI of a constant declaration in L , or (ii) even if it is, that constant may be declared without a definiens. The best solution for an elegant treatment of partial functions is to use option types $opt(t)$ akin to set types $set(t)$. However, for simplicity, we make $\llbracket - \rrbracket$ a partial function that is undefined whenever the interpretation of its argument runs into an undefined function application. This corresponds to the common concept of queries returning an error value.

The partial **functions** $type^M$ and def^M take the identifier of a constant declaration and return its type or definiens, respectively. They are undefined for other identifiers.

The partial function $infer^M(u, o)$ takes an object o and returns its dynamically inferred type. It is undefined if o is ill-typed. Since MMT does not commit to a type system, the argument u must identify the type system (which is represented as an MMT theory itself). If O is a binding object of the form $OMBIND(OMS(\mathbf{free}), \Gamma, o')$, the type of o' is inferred in context Γ .

arg_p is a family of function symbols indexed by a natural number p . p indicates the position of a direct subobject (usually an argument), and $arg_p^M(o)$ is the subobject of o at position p . In particular, $arg_i^M(OMA(f, a_1, \dots, a_n)) = a_i$. Note that arbitrary subobjects can be retrieved by iterating arg_p . Similarly, $subobj^M(o, h)$ is the set of all subobjects of o whose head is the symbol with identifier h . In particular, the head of $OMA(OMS(h), a_1, \dots, a_n)$ is h . In both cases, we keep track of the free variables, e.g., $arg_2^M(OMBIND(b, \Gamma, o)) = OMBIND(OMS(\mathbf{free}), \Gamma, o)$ for $b \neq OMS(\mathbf{free})$.

$unify^M(O)$ performs an object query: It returns the set of all tuples $u * o * s$ where u is the MMT URI of a declaration in L that contains an object o that unifies with O using the substitution s . Here we use a purely syntactic definition for unifiability of OPENMATH objects.

$render^M(o, u)$ and $render^M(d, u)$ return the presentation markup dynamically computed by the MMT rendering engine. This is useful because the query and the rendering engine are often implemented on the same remote server. Therefore, it is reasonable to compute the rendering of the query results, if desired, as part of the query evaluation. Moreover, larger signatures might provide additional functions to further operate on the presentation markup. $render$ is overloaded because we can present both MMT declarations and MMT objects. In both cases, u is the MMT URI of the style providing the notations for the rendering.

The **predicate** symbol $occurs$ takes an object O and an identifier u , and returns true if u occurs in O .

Finally, we permit **literals**, i.e., arbitrary URIs and arbitrary OPENMATH objects may be used as nullary constants, which are interpreted as themselves (or as undefined if they are not in the universe). This is somewhat inelegant but necessary in practice to form interesting queries. A more sophisticated QMT

signature could use one function symbol for every OPENMATH object constructor instead of using OPENMATH literals.

Example 7. An MMT theory graph is the multigraph formed by using the theories as nodes and all theory morphisms between them as edges. The components of the theory graph can be retrieved with a few simple queries.

Firstly, the set of theories is retrieved simply using the query *theory*. Secondly, the theory morphisms are obtained by two different queries:

$$\begin{aligned} \text{views} & \quad \{v * x * y : v \in \text{view}, x \in \text{domain}(v), y \in \text{domain}(v)\} \\ \text{inclusions} & \quad \bigcup_{y \in \text{theory}} \{x * y : x \in \text{includes}^*(y)\} \end{aligned}$$

The first one returns all view identifiers with their domain and codomain. Here we use an extension of the replacement operator $\{- : -\}$ from Ex. 3 to multiple variables. It is straightforward to define in terms of the unary one. The second query returns all pairs of theories between which there is an inclusion morphism.

Example 8. Consider a constant identifier $\exists I$ for the introduction rule of the existential quantifier from the natural deduction calculus. It produces a constructive existence proof of $\exists x.P(x)$; it takes two arguments: a witness w , and a proof of $P(w)$. Moreover, consider a theorem with identifier u . Recall that using the Curry-Howard representation of proofs-as-objects, a theorem u is a constant, whose type is the asserted formula and whose definiens is the proof.

Then the following query retrieves all existential witnesses that come up in the proof of u :

$$\{arg_1(x) : x \in \text{subobj}(\text{def}(u), \exists I)\}$$

Here we have used the replacement operator introduced in Ex. 3.

Example 9 (Continuing Ex. 8). Note that when using $\exists I$, the proved formula P is present only implicitly as the type of the second argument of $\exists I$. If the type system is given by, for example, LF and type inference for LF is available, we can extend the query from Ex. 8 as follows:

$$\{arg_1(x) * \text{infer}(LF, arg_2(x)) : x \in \text{subobj}(\text{def}(u), \exists I)\}$$

This will retrieve all pairs (w, P) of witnesses and proved formulas that come up in the proof of u .

4 Implementation

We have implemented QMT as a part of the MMT API. The implementation includes a concrete XML syntax for queries and an integration with the MMT web server, via which the query engine is exposed to users. The server can run as a background service on a local machine as well as a dedicated remote server. Sources, binaries, and documentation are available at the project web site [\[Rab08\]](#).

The MMT API already implements the MMT ontology so that appropriate indices for the semantics of all concept and relation symbols are available. Indices scale well because they are written to the hard drive and cached to memory on demand. With two exceptions, the semantics of all function and predicate symbols is implemented by standard MMT API functions.

The semantics of *unify* is computed differently: A substitution tree index of the queries library is maintained separately by an installation of MathWebSearch [KS06]. Thus, QMT automatically inherits some heuristics of MathWebSearch, such as unification up to symmetry of certain relation symbols. MathWebSearch and query engine run on the same machine and communicate via HTTP.

Another subtlety is the semantics of *infer*. The MMT API provides a plugin interface, through which individual type systems can be registered; the first argument to *infer*^M is used to choose an applicable plugin. In particular, we provide a plugin for the logical framework LF [HHP93], which handles type inference for any type system that is formalized in LF; this covers all type systems defined in the LATIN library [CHK⁺11] and thus also applies to our imports of the Mizar [TB85] and TPTP libraries [SS98].

Query servers for individual libraries can be set up easily. In fact, because the MMT API abstracts from different backends, queries automatically return results from all libraries that are registered with a particular instance of the MMT API. This permits queries across libraries, which is particularly interesting if libraries share symbols. Shared symbols arise, for example, if both libraries use the standard OpenMath CDs where possible or if overlap between the libraries' underlying meta-languages is explicated in an integrating framework like the LATIN atlas [CHK⁺11].

Example 10. The LATIN library [CHK⁺11] consists of over 1000 highly modularized LF signatures and views between them, formalizing a variety of logics, type theories, set theories, and related formal systems. Validating the library and producing the index for the MMT ontology takes a few minutes with typical desktop hardware; reading the index into memory takes a few seconds. Typical queries as given in this paper are evaluated within seconds.

As an extreme example, consider the query $Q = \text{Declares}(\text{theory})$. It returns in less than a second the about 2000 identifiers that are declared in any theory. The query $\bigcup_{x \in Q} \{x * \text{type}(x)\}$ returns the same number of results but pairs every declaration with its type. This requires the query engine to read the types of all declarations (as opposed to only their identifiers). If none of these are cached in memory yet, the evaluation takes about 4 minutes.

5 Conclusion and Future Work

We have introduced a simple, expressive query language for mathematical theories (QMT) that combines features of compositional, property, and object query languages. QMT is implemented on top of the MMT API; that provides any library that is serialized as MMT content markup with a scalable, versatile querying engine out of the box. As both MMT and its implementation are designed to

admit natural representations of any declarative language, QMT can be readily applied to many libraries including, e.g., those written in Twelf, Mizar, or TPTP.

Our presentation focused on querying *formal* mathematical libraries. This matches our primary motivation but is neither a theoretical nor a practical restriction. For example, it is straightforward to add a base type for presentation MathML and some functions for it. MathWebSearch can be easily generalized to permit unification queries on presentation markup. This also permits queries that mix content and presentation markup, or content queries that find presentation results. Moreover, for presentation markup that is generated from content markup, it is easy to add a function that returns the corresponding content item so that queries can jump back and forth between them.

Similarly, we can give a QMT signature with base types for authors and documents (papers, book chapters, etc.) as well as relations like `author-of` and `cites`. It is easy to generate the necessary indices from existing databases and to reuse our implementation for them. Moreover, with a relation `mentions` between papers and the type *id* of mathematical concepts, we can combine content and narrative aspects in queries. An index for the `mentions` relation is of course harder to obtain, which underscores the desirability of mathematical documents that are annotated with content URIs.

References

- ADL12. Aspinall, D., Denney, E., Lüth, C.: Querying Proofs. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18 2012. LNCS, vol. 7180, pp. 92–106. Springer, Heidelberg (2012)
- ANS03. ANSI/ISO/IEC. 9075:2003, Database Language SQL (2003)
- AS04. Asperti, A., Selmi, M.: Efficient Retrieval of Mathematical Statements. In: Asperti, A., Bancerek, G., Trybulec, A. (eds.) MKM 2004. LNCS, vol. 3119, pp. 17–31. Springer, Heidelberg (2004)
- AY08. Altamimi, M., Youssef, A.: A Math Query Language with an Expanded Set of Wildcards. *Mathematics in Computer Science* 2, 305–331 (2008)
- BCC⁺04. Buswell, S., Caprotti, O., Carlisle, D., Dewar, M., Gaetano, M., Kohlhase, M.: The Open Math Standard, Version 2.0. Technical report. The Open Math Society (2004), <http://www.openmath.org/standard/om20>
- BR03. Bancerek, G., Rudnicki, P.: Information Retrieval in MML. In: Asperti, A., Buchberger, B., Davenport, J.H. (eds.) MKM 2003. LNCS, vol. 2594, pp. 119–132. Springer, Heidelberg (2003)
- CHK⁺11. Codescu, M., Horozal, F., Kohlhase, M., Mossakowski, T., Rabe, F.: Project Abstract: Logic Atlas and Integrator (LATIN). In: Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.) *Calculemus/MKM 2011*. LNCS, vol. 6824, pp. 289–291. Springer, Heidelberg (2011)
- GC03. Guidi, F., Sacerdoti Coen, C.: Querying Distributed Digital Libraries of Mathematics. In: Hardin, T., Rioboo, R. (eds.) *Proceedings of Calculemus*, pp. 17–30 (2003)
- HHP93. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. *Journal of the Association for Computing Machinery* 40(1), 143–184 (1993)
- Koh06. Kohlhase, M.: OMDoc – An Open Markup Format for Mathematical Documents [version 1.2]. LNCS (LNAI), vol. 4180. Springer, Heidelberg (2006)

- KRZ10. Kohlhase, M., Rabe, F., Zholudev, V.: Towards MKM in the Large: Modular Representation and Scalable Software Architecture. In: Autexier, S., Calmet, J., Delahaye, D., Ion, P., Rideau, L., Rioboo, R., Sexton, A. (eds.) AISC 2010. LNCS, vol. 6167, pp. 370–384. Springer, Heidelberg (2010)
- KŠ06. Kohlhase, M., Sucan, I.: A Search Engine for Mathematical Formulae. In: Calmet, J., Ida, T., Wang, D. (eds.) AISC 2006. LNCS (LNAI), vol. 4120, pp. 241–253. Springer, Heidelberg (2006)
- LM06. Libbrecht, P., Melis, E.: Methods to Access and Retrieve Mathematical Content in ACTIVEMATH. In: Iglesias, A., Takayama, N. (eds.) ICMS 2006. LNCS, vol. 4151, pp. 331–342. Springer, Heidelberg (2006)
- MG08. Mišutka, J., Galamboš, L.: Extending full text search engine for mathematical content. In: Sojka, P. (ed.) Towards a Digital Mathematics Library, pp. 55–67 (2008)
- MM06. Munavalli, R., Miner, R.: MathFind: a math-aware search engine. In: Efthimiadis, E., Dumais, S., Hawking, D., Järvelin, K. (eds.) International ACM SIGIR Conference on Research and Development in Information Retrieval, p. 735. ACM (2006)
- MY03. Miller, B., Youssef, A.: Technical Aspects of the Digital Library of Mathematical Functions. *Annals of Mathematics and Artificial Intelligence* 38(1-3), 121–136 (2003)
- Rab08. Rabe, F.: The MMT System (2008), <https://trac.kwarc.info/MMT/>
- RK11. Rabe, F., Kohlhase, M.: A Scalable Module System (2011), <http://arxiv.org/abs/1105.0548>
- SL11. Sojka, P., Líška, M.: Indexing and Searching Mathematics in Digital Libraries - Architecture, Design and Scalability Issues. In: Davenport, J., Farmer, W., Urban, J., Rabe, F. (eds.) *Calculus/MKM 2011*. LNCS, vol. 6824, pp. 228–243. Springer, Heidelberg (2011)
- SS98. Sutcliffe, G., Suttner, C.: The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning* 21(2), 177–203 (1998)
- TB85. Trybulec, A., Blair, H.: Computer Assisted Reasoning with MIZAR. In: Joshi, A. (ed.) *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pp. 26–28 (1985)
- Urb06. Urban, J.: MOMM - Fast Interreduction and Retrieval in Large Libraries of Formalized Mathematics. *International Journal on Artificial Intelligence Tools* 15(1), 109–130 (2006)
- W3C03. W3C. Mathematical Markup Language (MathML) Version 2.0., 2nd edn. (2003), <http://www.w3.org/TR/MathML2>
- W3C07. W3C. XQuery 1.0: An XML Query Language (2007), <http://www.w3.org/TR/xquery/>
- W3C08. W3C. SPARQL Query Language for RDF (2008), <http://www.w3.org/TR/rdf-sparql-query/>
- ZK09. Zholudev, V., Kohlhase, M.: TNTBase: a Versioned Storage for XML. In: *Proceedings of Balisage: The Markup Conference 2009*. Balisage Series on Markup Technologies, vol. 3, Mulberry Technologies, Inc. (2009)

Abramowitz and Stegun – A Resource for Mathematical Document Analysis

Alan P. Sexton

School of Computer Science
University of Birmingham, UK
a.p.sexton@cs.bham.ac.uk

Abstract. In spite of advances in the state of the art of analysis of mathematical and scientific documents, the field is significantly hampered by the lack of large open and copyright free resources for research on and cross evaluation of different algorithms, tools and systems.

To address this deficiency, we have produced a new, high quality scan of Abramowitz and Stegun’s Handbook of Mathematical Functions and made it available on our web site. This text is fully copyright free and hence publicly and freely available for all purposes, including document analysis research. Its history and the respect in which scientists have held the book make it an authoritative source for many types of mathematical expressions, diagrams and tables.

The difficulty of building an initial working document analysis system is a significant barrier to entry to this research field. To reduce that barrier, we have added intermediate results of such a system to the web site, so that research groups can proceed on research challenges of interest to them without having to implement the full tool chain themselves. These intermediate results include the full collection of connected components, with location information, from the text, a set of geometric moments and invariants for each connected component, and segmented images for all plots.

1 Introduction

Reliable, high quality tools for optical analysis and recognition of mathematical documents would be of great value in mathematical knowledge management. Such tools would enable large scale, low cost capturing of mathematical knowledge both from scanned images of documents and from electronic versions such as the usual PDF formatted output from \LaTeX , which does not include the semantic enrichments necessary for a knowledge-oriented representation of mathematics.

In spite of advances in the state of the art of analysis of mathematical and scientific documents [21], the field is significantly hampered by the lack of suitable large, open and copyright free document sets to serve as ground truth sets and as data sets for testing and cross evaluation of different algorithms, tools and systems.

A *data set*, in this context, is usually a collection of page images from mathematical documents, although they can be image clips of formulae from those pages, or collections of character, symbol or diagram images.

A *ground truth set* is an input data set together with validated correct recognition results for the input data set. These correct recognition results are normally produced manually or by automatic recognition followed by manual correction. In particular they include full character and formula identification in spite of problems such as touching or broken characters — even if these problems are beyond the current state of the art of document analysis to resolve. As such it provides an ideal recognition result by which document analysis systems can be trained and against which they can be evaluated.

The most significant data sets for optical analysis of printed mathematical documents is the series of data sets from the Infty project [8,16,18]:

- InftyCDB-1: 476 pages of ground truth text from 30 mathematics articles in English with 688,580 character samples, 108,914 words and 21,056 mathematical expressions.
- InftyCDB-2: has the same structure as InftyCDB-1. It has 662,142 characters from English articles, 37,439 from French, and 77,812 from German.
- InftyCDB-3: is divided into two data sets. InftyCDB-3-A contains 188,752 characters. InftyCDB-3-B contains 70,637 characters. Word and mathematical expression structure is not included.
- InftyMDB-1: contains 4,400 ground truth mathematical formulae from 32 articles, which are mostly the same articles as in InftyCDB-1.

While an excellent resource for the mathematical OCR community, and especially useful for optical character and mathematical formula recognition training purposes, the one drawback of these data sets is that the original articles are not copyright free. The articles can not be distributed with the data sets and the data sets are not allowed to contain sufficient information about the location of characters so that the articles could be reconstructed from the data. Hence one can not easily test and compare systems on the original articles while using the ground truth for evaluation purposes.

Another data set is UW-III [14], the technical document ground truth data set from the University of Washington, containing 25 pages of ground truth mathematics. This data set is not free, currently costing \$200 plus shipping.

Garain and Chaudhuri [9,10] discusses their proposed ground truth corpus of 400 real and synthetic mathematical document images. However their available data set [9] is of 102 isolated expression images and 5 clips from mathematical document pages, all with ground truth information.

Ashida et al. [2] produced a ground truth data set of symbols from 1400 pages and formulae from 700 pages of mathematics (taken from *Archiv der Mathematik* and *Commentarii Mathematici Helvetici*). Unfortunately, this data set is not available for copyright reasons.

Building a ground truth for a significant number of mathematical document pages is a very expensive and labour intensive project. It is particularly

unfortunate that, to date, no entirely copyright and cost free data set, for which the original documents are also copyright and cost free, has yet been made available.

In the remainder of this paper, we discuss a project to make available a very large, high quality, copyright and cost free data set for optical analysis of mathematical documents which, although not yet ground truthed, has been prepared to simplify processing and support future community based ground truthing.

2 Abramowitz and Stegun

Abramowitz and Stegun [1], or A&S, is an encyclopedia of mathematical functions. As it was published by the United States Government Printing Office, it is copyright free and hence fully available for researchers (and anyone else) to scan, report on and make their results freely available. Its history [3] and the respect in which scientists have held the book make it an authoritative source for many types of expressions, diagrams and tables, as witnessed by the article on the book in Wikipedia:

“Abramowitz and Stegun is the informal name of a mathematical reference work edited by Milton Abramowitz and Irene Stegun of the U.S. National Bureau of Standards (now the National Institute of Standards and Technology). Its full title is Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.

Since it was first published in 1964, the 1046 page Handbook has been one of the most comprehensive sources of information on special functions, containing definitions, identities, approximations, plots, and tables of values of numerous functions used in virtually all fields of applied mathematics. The notation used in the Handbook is the de facto standard for much of applied mathematics today.”¹

The fact that A&S was printed pre- \TeX means that it can help researchers to avoid over-tuning their systems to the much more readily available \TeX / \LaTeX sourced documents. For \TeX documents, the availability of the ArXiv [5] with its very large collection of \TeX based documents with full sources, makes alternative methods of obtaining data sets for OCR much more productive. Indeed, the ArXiv is already being data mined for the purposes of analysing semantic content [15]

One could argue that the ArXiv is such a large and rich resource that there is no need for a resource such as the one discussed here based on A&S. We respond to this argument as follows:

1. Fonts and typesetting are different enough between \TeX and non- \TeX based documents to significantly effect recognition. There is an enormous amount of mathematics in non- \TeX based material still available only as bitmap images. Using only \TeX based documents to train and test document analysis systems could only handicap those systems in just the same way that not

¹ Taken from: http://en.wikipedia.org/wiki/Abramowitz_and_Stegun

using any \TeX based document data sets would: both \TeX based and non- \TeX based data sets are necessary.

2. It is significantly easier to produce a ground truth automatically when \TeX sources, or PDF produced from \TeX , are available. However, this does not cover the significant range of problems that earlier printing technologies introduce, such as the number and type of broken and connected characters, the level of ink bleed, problems caused by manual, rather than automatic typesetting etc. Modern printing followed by re-scanning produces artificial data sets that suffer from a much restricted and biased set of problems relative to those that occur in the wild. Hence it is valuable to have a data set such as this one that results from a true, non-digital printing process in addition to the data sets that can be built from collections such as the ArXiv.
3. A&S is of interest in itself. There is interest in matching the equations in A&S with those in the DLMF [12], it contains sections on probability that are not included in the DLMF and the tables in A&S, due to the care and rigour with which they were produced [3], are of interest in themselves if they could be accurately recognised.

The nature of A&S makes it a particular challenge for mathematical document analysis systems. Its 1060 pages contains a very high density of mathematical formulae (c.f. Figure 1). It has a complex layout that makes segmentation difficult — relatively few and short plain text passages, a two column default layout with lines usually, but not always, separating them and with frequent switches to single column layouts.

$$f(s) = \mathcal{L}\{F(t)\} = \int_0^{\infty} e^{-st} F(t) dt$$

(a) Equation 1

$$p(n) = \frac{1}{\pi\sqrt{2}} \sum_{k=1}^{\infty} \sqrt{k} A_k(n) \frac{d}{dn} \frac{\sinh\left\{\frac{\pi}{k}\sqrt{\frac{2}{3}}\sqrt{n-\frac{1}{24}}\right\}}{\sqrt{n-\frac{1}{24}}}$$

(b) Equation 2

Fig. 1. Example Equations from A&S

A&S also has a very large numbers of tables (c.f. Figure 2), both of numbers and of mathematical formulae. Many of these tables span multiple pages, most without any inter-cell lines but many with a complex line structure and a significant number rotated to landscape format. Some of the tables have been printed with approximately 5pt font sizes. Often the layout is spatially squeezed with obvious manual adjustments.

n	$\operatorname{erf} z_n=0$		$z_n=x_n+iy_n$		y_n
	x_n	y_n	x_n	y_n	
1	1.45061 616	1.88094 300	6	4.15899 840	4.43557 144
2	2.24465 928	2.61657 514	7	4.51631 940	4.78044 764
3	2.83974 105	3.17562 810	8	4.84797 031	5.10158 804
4	3.33546 074	3.64617 438	9	5.15876 791	5.40333 264
5	3.76900 557	4.06069 723	10	5.45219 220	5.68883 744

$\operatorname{erf} z_n = \operatorname{erf}(-z_n) = \operatorname{erf} \bar{z}_n = \operatorname{erf}(-\bar{z}_n) = 0$

Fig. 2. Example Table from A&S

There are a large number of often complex plots (c.f. Figure 3).

The printing suffered from some problems in that there are a large number of characters that are clearly broken, and other characters that are touching, as well as reliably reproduced dirt or marks on the pages. These faults do not diminish the readability of the text to a human, but cause issues for OCR software.

3 Rescan and Analysis

A digital capture of a new copy of A&S was carried out. It was scanned at 600dpi to 8 bit grey scale TIFF images. The files contain all pages in the book, including the front and back matter and blank pages separating sections. The latter were included so that the relationship between TIFF/PDF file page numbers and book page numbers would remain as simple as possible.

The original printing process for the book was such that there are a significant number of printing flaws in the original book. Many flaws in the scanned images are faithful reproductions of these printing flaws rather than artifacts of the scanning process. In particular, most pages of the book have some slight skew — up to 1.35° in the worst cases. While the scanning process undoubtedly introduced some level of skew, most of the skew appears in the original book.

Post-scanning processing of the images was carried out to deskew, binarise and remove any connected components smaller than the size of the smallest correct dot.

The deskewing was carried out automatically based on a projection profile approach and, although it is by no means perfect, it has reduced the skew in all cases. The resulting processed images, at 600dpi and reduced to 300dpi, are available at the project web site.

3.1 Connected Components and Geometric Moments

A connected component in a monochrome image is a maximal set of connected foreground pixels. Extracting them from an image is typically the first step of an OCR system after image pre-processing (binarisation, noise removal etc.). All connected components from the 600dpi monochrome images of A&S were extracted and have been made available. Although each connected component could be stored as a g4 compressed monochrome TIFF image, we have stored them as 8 bit grey scale TIFF images with an alpha, or transparency, channel

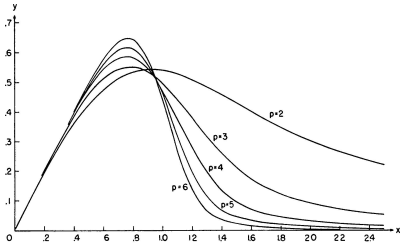


FIGURE 7.2. $y = e^{-z^p} \int_0^z e^{t^p} dt$.
 $p = 2(1)6$

(a) Line Plot 1

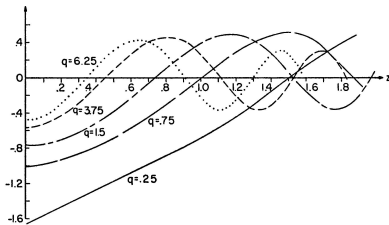


FIGURE 20.13. Radial Mathieu Function of the Second Kind.

(b) Line Plot 2

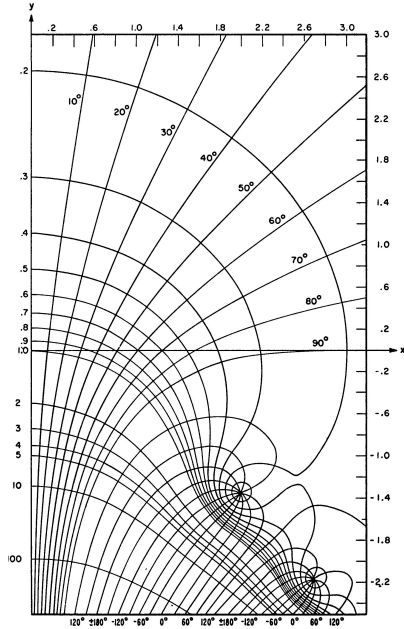


FIGURE 7.3. Altitude Chart of $w(z)$.

(c) Chart

Fig. 3. Example Plots from A&S

and deflate compression. The foreground colour for these images is black, as expected. However the background colour has been set to fully transparent white. The result is that the original page image can be easily reconstructed for test purposes by drawing all the connected component images of a page in the correct location on a white page, The transparent background of the images ensures that no individual image masks any part of another image just because the bounding boxes of the images overlap. The resulting image files are not significantly larger than the monochrome versions and can easily be flattened to true monochrome if so desired.

There were 2,672,788 connected components extracted in total and the number of connected components per page ranges from 270 to 12824, with an average of 2572.

A data file, in comma separated value (CSV) format, was prepared that identifies the correct page, location and bounding box size of all extracted connected components.

One of the most historically popular approaches to optical printed character recognition is based on geometric moments and moment invariants [6,13,20]. In order to lower the barrier to entry for students and groups to mathematical document analysis research, we have pre-calculated and provided a set of features based on these moments for each connected component and included them in the CSV data file. The features included are

- An aspect ratio feature:

$$\frac{1}{2} + \frac{h - w}{2 \max(h, w)}$$

where h, w are the height and width respectively of the bounding box of the connected component. This returns a number between 0 and 1.

- m_{00} : the $(0, 0)$ geometric moment, which corresponds to the number of foreground pixels in the connected component.
- $\eta_{20}, \eta_{11}, \eta_{02}, \eta_{30}, \eta_{21}, \eta_{12}, \eta_{03}$: all the second and third order normalised central geometric moments (by definition, $\eta_{00} = 1$ and $\eta_{01} = \eta_{10} = 0$). These are translation and scale invariant, but not rotation or general affine transform invariant.
- I_1, I_2, \dots, I_8 : the extended set of Hu invariants [11]. Hu defined 7 rotation invariant moments, I_1, \dots, I_7 . However, this set was shown by Flusser [6] to be neither independent (in particular, I_3 can be derived from I_4, I_5 and I_7 and therefore can be omitted without loss of recognition power) nor complete, hence the addition of the I_8 moment.

In the form that these features are provided, they can easily be used as the basis for any number of pattern classification approaches such as a metric space based k-nearest neighbour or cosine similarity [4, 19]. However, the ranges of the values are such that some features will overwhelm others without further scaling. For this reason we include the ranges for all features in Table 1.

Table 1. Numeric Feature Ranges

Feature	Min	Max
aspect	0.0006785795	0.9989733
m_{00}	6	2187349
η_{20}	$5.212947e - 05$	64.39165
η_{11}	-10.16923	12.31475
η_{02}	0.0001275819	132.2743
η_{30}	-135.2704	158.1855
η_{21}	-27.66605	28.00989
η_{12}	-39.19975	15.09384
η_{03}	-347.5878	135.2028
I_1	0.15625	132.2746
I_2	0	17496.43
I_3	0	120749
I_4	0	120845
I_5	-524301100	14597710000
I_6	-28445.19	15781260
I_7	-172762300	548055900
I_8	-148817.6	281045.5

4 Special Clips

A particular asset of A&S is its wealth of plots. There is a large range of plots encompassing simple line plots, contour maps and branch cut plots, as shown in Figure 3. There is interest in work on analysing and recognising such plots [7,17]. To support that work we have manually clipped and extracted all 399 plots from A&S and provided them, with location and bounding box information, with the resources available from our web site. Together with the feature information about the connected components in A&S, this provides a low barrier to entry for research in this area.

5 Conclusions

The main aim of the work has been to provide a much needed resource of high quality for the mathematical document analysis community. While other data sets provide the very important aspect of ground truthing, none of them are, to date, fully open and free. The data set reported on here, while not provided with ground truth data, is fully open and free, and future work to develop a ground truth based on automatic recognition and community based crowd-sourced correction is planned.

A further aim has been to lower the barriers to entry to research for students interested in this area. Without such a data set, and partially processed initial analyses, an undergraduate or MSc student has too little time, after completing the software for initial processing, to pursue the more interesting goals of symbol recognition, formula recognition and layout analysis. It is hoped that with this data, this will no longer be the case.

We plan on adding full character recognition information to this data set in the near future and invite contributions from other research groups to enhance the data set.

The project web site, and all data sets and images from the project, is available at <http://www.cs.bham.ac.uk/~aps/research/projects/as/>.

Because of their size (approximately 19GB), the full, *losslessly compressed* set of original, 600 dots per inch, grey scale scanned images without any image processing applied are not accessible directly from the web site, although they are available from the author on request.

A version of both the original grey scale images and the deskewed grey scale images in TIFF format with lossy JPEG compression at a compression level of 60% is available from the web site. Each of these sets comes to 2.5GB and they are actually quite usable for document analysis purposes.

It is expected that these data sets may be useful for research on binarisation, noise reduction, deskewing, or grey scale optical character recognition.

Acknowledgements. We thank Bruce Miller of the National Institute of Standards and Technology, for providing a clean new copy of A&S for scanning.

We thank Bruno Voisin, of the Laboratory of Geophysical and Industrial Flows (LEGI), Grenoble, France, who allowed the use of his code to create PDF bookmarks for A&S as a basis for the bookmarks in the PDFs on the project web site.

References

1. Abramowitz, M., Stegun, I.A.: Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables. US Government Printing Office, Washington, 10th printing, with corrections (December 1972)
2. Ashida, K., Okamoto, M., Imai, H., Nakatsuka, T.: Performance evaluation of a mathematical formula recognition system with a large scale of printed formula images. In: International Workshop on Document Image Analysis for Libraries, pp. 320–331 (2006), <http://doi.ieeecomputersociety.org/10.1109/DIAL.2006.30>
3. Boisvert, R.F., Lozier, D.W.: Handbook of mathematical functions. In: Lide, D.R. (ed.) A Century of Excellence in Measurements Standards and Technology, pp. 135–139. CRC Press (2001), <http://nvl.nist.gov/pub/nistpubs/sp958-lide/135-139.pdf>
4. Cheriet, M., Kharma, N., Liu, C.L., Suen, C.Y.: Character Recognition Systems — A Guide for Students and Practitioners. Wiley & Sons Ltd., Hoboken (2007)
5. Cornell University Library (2012), <http://www.arxiv.org>
6. Flusser, J., Suk, T., Zitová, B.: Moments and Moment Invariants in Pattern Recognition. Wiley & Sons Ltd., Chichester (2009)
7. Fuda, T., Omachi, S., Aso, H.: Recognition of line graph images in documents by tracing connected components. Trans. IEICE J86-D-II(6), 825–835 (2003)
8. Fujiyoshi, A., Suzuki, M., Uchida, S.: Verification of Mathematical Formulae Based on a Combination of Context-Free Grammar and Tree Grammar. In: Autexier, S., Campbell, J., Rubio, J., Sorge, V., Suzuki, M., Wiedijk, F. (eds.) AISC/Calculemus/MKM 2008. LNCS (LNAI), vol. 5144, pp. 415–429. Springer, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-85110-3_35
9. Garain, U., Chaudhuri, B.B.: Ground truth datasets of mathematics, <http://www.isical.ac.in/~utpal/resources.php>
10. Garain, U., Chaudhuri, B.B.: A corpus for OCR research on mathematical expressions. IJDAR 7(4), 241–259 (2005), <http://dx.doi.org/10.1007/s10032-004-0140-5>
11. Hu, M.K.: Visual pattern recognition by moment invariants. IRE Transactions on Information Theory 8(2), 179–187 (1962)
12. Miller, B.: Personal communication (2011)
13. Mukundan, R., Ramakrishnan, K.: Moment Functions in Image Analysis. World Scientific, Singapore (1998)
14. Phillips, I., Chanda, B., Haralick, R.: UW-III english/technical document image database. University of Washington (2000), <http://www.science.uva.nl/research/dlia/datasets/uwash3.html>
15. Stamerjohanns, H., Kohlhase, M.: Transforming the arXiv to XML. In: Autexier, S., Campbell, J., Rubio, J., Sorge, V., Suzuki, M., Wiedijk, F. (eds.) AISC/Calculemus/MKM 2008. LNCS (LNAI), vol. 5144, pp. 574–582. Springer, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-85110-3_46

16. Suzuki, M., Uchida, S., Nomura, A.: A ground-truthed mathematical character and symbol image database. In: Eighth International Conference on Document Analysis and Recognition (ICDAR 2005), pp. 675–679 (2005), <http://doi.ieeecomputersociety.org/10.1109/ICDAR.2005.14>
17. Takagi, N.: On consideration of a pattern recognition method for mathematical graphs with broken lines. In: International Workshop on Digitization and E-Inclusion in Mathematics and Science (DEIMS 2012), Tokyo, pp. 43–51 (2012)
18. The Infty Project: InftyCDB-1–3, InftyMDB-1 (2009), <http://www.inftyproject.org/en/database.html>
19. Theodoridis, S., Koutroumbas, K.: Pattern Recognition, 4th edn. Academic Press (2009)
20. Yampolskiy, R.: Feature Extraction Approaches for Optical Character Recognition. Briviba Scientific Press, Rochester (2007)
21. Zanibbi, R., Blostein, D.: Recognition and retrieval of mathematical expressions. International Journal on Document Analysis and Recognition, 1–27 (2012), <http://dx.doi.org/10.1007/s10032-011-0174-4>

Point-and-Write – Documenting Formal Mathematics by Reference^{*}

Carst Tankink¹, Christoph Lange^{2,3,4}, and Josef Urban¹

¹ Institute for Computing and Information Science, Radboud Universiteit, Nijmegen, The Netherlands

carst@cs.ru.nl, josef.urban@gmail.com

² FB 3, Universität Bremen, Germany

ch.lange@jacobs-university.de

³ Computer Science, Jacobs University Bremen, Germany

⁴ School of Computer Science, University of Birmingham, UK

Abstract. This paper describes the design and implementation of mechanisms for light-weight inclusion of formal mathematics in informal mathematical writings, particularly in a Web-based setting. This is conceptually done in three stages: (i) by choosing a suitable representation layer (based on RDF) for encoding the information about available resources of formal mathematics, (ii) by exporting this information from formal libraries, and (iii) by providing syntax and implementation for including formal mathematics in informal writings.

We describe the use case of an author referring to formal text from an informal narrative, and discuss design choices entailed by this use case. Furthermore, we describe an implementation of the use case within the Agora prototype: a Wiki for collaborating on formalized mathematics.

1 Introduction

Formal, computer-verified, mathematics has been informally discussed and written about for some fifty years: on dedicated mailing lists [10,19,13], in conference and journal articles, online manuals, tutorials and courses, and in community Wikis [9,20] and blogs [22].

In such informal writings, it is common to include and mix formal definitions, theorems, proofs and their outlines, and sometimes whole sections of formal articles. Such formal “islands” in a text do not have to follow any particular logical order, and can mix content from different articles, libraries, and even content based on different proof assistants. In this respect, the collection of such formal fragments in a particular text is often *informal*, because the fragments do not have to share and form a unifiable, linear, and complete *formal context*.

* The first and third author were funded by the NWO project “MathWiki”. The second author was supported by DFG Project I1-[OntoSpace] of SFB/TR 8 “Spatial Cognition” and EPSRC grant EP/J007498/1”. The final publication is available at <http://www.springerlink.com>.

In a Web setting, such pieces of formal code can however be equipped with semantic and presentation functions that make formal mathematics attractive and unique. Such functions range from “passive” markup, like (hyper)linking symbols to their precise definitions in HTML-ized formal libraries, detailed and layered explanations of implicit parts of reasoning (goals, types, subproofs, etc.), to more “active”, like direct editing, re-verification, and HTML-ization of the underlying formal fragment in its proper context, and using the formal code for querying semantic search engines and automated reasoning tools [25].

In this paper, we describe, and support, a use case of an author writing such an informal text: she gives references (**points**) to (fragments of) formalization on the Web, and then describes (**writes** about) them in a natural language narrative, documenting the formal islands. This use case is described in Section 2.

We support this use case in a light-weight manner, based on HTML presentations of formal mathematics. The author can write pointers to formal objects in a special syntax (described in Section 3), which get resolved to the objects when rendering the narrative. To follow the pointers, our tools equip HTML pages for formalizations with annotations describing *what* a particular HTML fragment represents (Section 5). The annotations are drawn from suitable RDF vocabularies¹ described in Section 4.

We show an implementation of the mechanisms in the Agora prototype² described in Section 6. The actual *rendering* of the final page with its inclusions give rise to several issues that we do not consider here: we discuss some issues and how we handle them in our implementation, but these decisions were not made systematically: we focus here on the author’s use case of writing the references, and provide the prototype as a proof of existence.

This paper does not contain a dedicated related work section: to our knowledge, there is no system that provides similar functionality: there are alternatives for including formal text, as well as alternative syntaxes, which we will compare with our approach in the relevant sections. Additionally, the MoWGLI project developed some techniques for rendering formal proofs with informal narratives, which are compared to in Section 6.

2 Describing *and* Including Formal Text

The techniques described in this paper are mainly driven by a single use case, that of an author writing a description (a “narrative”) of a development in formal, computer-verified, mathematics. In this work, we assume that the author writes this narrative for publication in a Wiki, although the use case could also be applied for more traditional authoring, in a language like L^AT_EX.

¹ The RDF data model (Resource Description Framework) essentially allows for identifying any thing (“resource”) of interest by a URI, giving it a type, attaching data to it, and representing its relations to other resources [1].

² <http://mws.cs.ru.nl/agora/>

2.1 Use Case

While writing a natural language narrative, the author will eventually want to include snippets of formal code: for example to illustrate a particular implementation technique or to compare a formalization approach with a different one, possibly in a different formal language. An advanced example is Section 5.3 of [6] rendered in Agora.³

Because we allow for including formal text from the Web, there is a wrinkle we have to iron out when supporting this use case, but before we get to that, we describe typical steps an author can carry out while executing the use case:

Formalization. An author works on a formalization effort in some system and puts (parts of) this formalization on the Web, preferably on the Wiki. We will refer to the results of these efforts as **source texts**.

Natural Language Description. Sometime before, during or after the formalization, the author gives a natural language account of the effort on the Wiki. As mentioned, we assume she writes this in a markup language suitable for Wikis, extended with facilities for writing mathematical formulae.⁴ We refer to the resulting description as a **narrative**.

Including Formalizations into the Narrative. In the natural language description, the author includes some of the formal artifacts of her effort, and possibly some of the formalizations by other authors. These inclusions need to look attractive (by being marked up) and should not be changed from the source: the source represents a verifiable piece of mathematics, and a reader should be able to ascertain himself that nothing was lost in transition.

These steps are not necessarily carried out in order, and can be carried out by different authors or iteratively. In particular, the formal text included in the narrative does not have to originate from the author or her collaborators, but could be from a development that serves as competition or inspiration.

The end results of the workflow are pages like the one shown (in part) in Figure 1: it includes a narrative written in natural language (including hyperlinks and markup of formulae) and displays formal definitions marked up as code.

Because the source texts are stored on the Web, we consider their content to be *fluid*: subject to change at any particular moment, but not under control of the author. This implies that the mechanism for including formal text should be robust against as much change as possible.

To determine how we can support this workflow, we first survey the existing methods that are suitable for including formal mathematics.

2.2 Alternatives for Inclusion

Typical options for including formal mathematics—or any other type of code—, when working with a document authoring tool like L^AT_EX include:

³ http://mws.cs.ru.nl/agora/cicm_sandbox/CCL/

⁴ The specifics of suitable languages for writing in Wikis for formal mathematics are not a subject of this paper; we refer to [18] for an overview.

Coq

In Coq's *standard library*, the binomial coefficient is *defined* computationally, as:

```
Definition C (n q:nat) : R :=
  INR (fact n) / (INR (fact q) * INR (fact (n - q))).
```

This definition matches the formula $\frac{n!}{k!(n-k)!}$ for computing the value of the coefficient.

Fig. 1. Example of informal narrative with formal snippets

1. **Referral:** place the code on some Web page and refer readers to that page from the document (by giving the URL),
2. **Inclusion:** include and format the source code files as listings, e.g. using the L^AT_EX package listings [12],
3. **Literate Proving:** the more extreme variant of (2): write the article in a literate style, and extract both formal code and marked up text from it,
4. **Copy-Paste:** manually copy-paste the code into the document.

All of these options have their own problems for our use case.

1. Referral collides with the desire for *juxtaposability* [7]: a reader should not have to switch between pages to look at the referred code and the text that refers to it. Instead, he should be able to read the island within the context of the narrative.
2. We certainly want the author to be able to include code, but most of the tools only allow her to refer to the code by *location*, instead of a more semantic means: she can give a range of lines (or character offsets) in a file, but cannot write “include the Fundamental Theorem of Algebra, and its proof”.
3. Literate proving [8] is a way to tackle code inclusion, but it does not solve the use case: it requires the author to shift her methods from writing code and article separately to writing both aspects interleaved.

It also does not allow an author to include existing external code for citation (without copy-pasting) and does not allow her to write a document including only snippets of formal code. These cases can arise where a lot of setup and auxiliary lemmas are necessary for formalizing a theorem, but only the theorem itself is the main focus of a paper. Typical literate programming setups provide mechanisms for hiding code fragments, but we prefer to take an inclusive instead of an exclusive view on the authoring process: the former seems to be more in line with actual practices in the interactive theorem proving community (see, for example, the proceedings of the Interactive Theorem Proving conference [26]).

4. Copy-pasting code has the traditional problem of maintaining consistency: if the source file is changed, the citation should change as well. On the positive side, it does not require much effort to implement, apart from adding facilities for marking up code, which can be reused for in-line (new) code. To make the implementation threshold even lower, the listings \LaTeX package previously mentioned also supports marking up copy-pasted code.

The shortcomings of these methods mean we need to design a system providing the following facilities:

Requirement 1. *A syntax for writing, in a natural language document, references to parts of a formal text, possibly outside of the referring text, and a mechanism for including the referred objects verbatim in a rendered version of the natural language text.*

Requirement 2. *A method for annotating parts of formal texts, so they can be referenced by narratives.*

The rest of this paper gives our approach to these two problems, demonstrating how they interact, and gives a tour of our working implementation⁵

3 Syntax for Referring

We will first focus on *how* the author can write references to formal content. Below, we discuss considerations that guided the design of this syntax. The considerations are partially based on the goals for a common Wiki syntax [21].

3.1 Requirements on Syntax

Simple. To encourage its use, the syntax should not be too elaborate. An example of a short enough syntax is the hyperlink syntax in most Wiki systems: only four characters surrounding the link, [[and]].

Collision Free. The syntax should not easily be ‘mistyped’: it should not be part of the syntax already used for markup, and not likely used in a natural language narrative.

Readable. It should be recognizable in the source of the narrative, to support authors in learning a new syntax and making the source readable.

Familiar. We do not intend to reinvent the wheel, but want to adapt existing syntax to suit our needs. This also keeps the syntax readable: when the base syntax is already known to an author, it should be clear to her how this syntax works in the context of referring to formal text. Because keeping things similar but not completely equal could cause confusion, it also requires us not to deviate the behavior too much from the original syntax.

⁵ http://mws.cs.ru.nl/agora/cicm_sandbox

In resolving these requirements on the syntax, we need to consider the context in which it will be used. In our proposed use case, the syntax will be used within a Wiki, so we prefer a syntax that fits with the markup families used for Wiki systems. It should be possible to extend a different markup language (for example, \LaTeX or literate comments for a formal system) with the reference syntax, but this requires reconsidering the decisions we make here.

Considering that we want to base the reference syntax on existing mechanisms (in line with the familiarity requirement), there are three basic options to use as a basis: import statements like used in \LaTeX (or, programming and formal languages), Wiki-style hyperlinks, and Isabelle/Isar's antiquotation syntax.

Each of these is considered in the rest of this section, and tested against the requirements stated before.

\LaTeX -Style Include Statements. The purpose of these statements in a \LaTeX document is to include the content of a file in another, before rendering the containing file. The command does not allow inclusion of file fragments, but could be modified to allow this. As a concrete proof of existence, the listings package mentioned earlier has the option to include file fragments by giving a line offset, but not a pointer to an object. The statements should be recognizable by \LaTeX users or users of a formal language that uses inclusions, but the statements are rather long: if the author wants to use them more often, it might become tedious to write.

The MediaWiki engine has a similar syntax for including entire pages⁶. To include fragments of pages, however, one either needs to factor out these fragments of the source text and include them both in the source text and the referring text, or mark fragments of the source page which will be included. Both do not give the author of a narrative fine-grained control over inclusion.

An extension to these inclusions⁷ allows inclusion of sections. This mechanism is a valid option for adaption, but if we would want to support informal inclusion at some point, it would be difficult to distinguish it, at the source level, from an inclusion of a formal fragment.

Wiki-Style Hyperlinks. These cross-reference statements are not hard to learn and short, but also using them for inclusion can overload the author's understanding of the markup commands: if she already knows how to use hyperlinks, she needs to learn how to write and recognize links that include formal objects.

Antiquotations. Isabelle/Isar [28] uses antiquotations to allow the author of Isar proof documents to write natural-language, marked-up snippets in a formal document (the 'quotation' from formal to informal), while including formal content in these snippets (the 'antiquotation' of informal back to formal): these antiquotations are written `@{type [options] syntax }`, where `type` declares what kind of syntax the formal system can expect, the `syntax` specifying the formal content,

⁶ <http://www.mediawiki.org/wiki/Transclusion>

⁷ http://www.mediawiki.org/wiki/Extension:Labeled_Section_Transclusion

and the options defining how the results should be rendered. The formal system interprets these snippets and reinserts the results into the marked-up text.

For example, the antiquotation `@{term [show_types] "% x y. x"}` would ask Isabelle to type-check the term $\lambda x y.x$ (% is Isabelle's ASCII shorthand for λ) in the context where it appears and reinserts the term annotated with its type: it inserts the output $\lambda(x::'a) y::'b. x$.

Another example is `@{thm foo}` which inserts the statement of the theorem labeled `foo` in the marked up text. The syntax also provides an option to insert the label `foo`, which makes sure that it points to a correct theorem.

3.2 Resulting Syntax

From the options listed above, the antiquotation mechanism is closest to what we want: it allows the inclusion of formal text within an informal environment, relying on an external (formal) system to provide the final rendering. There are some differences in the approaches that require some further consideration.

Context. In Isar, the informal fragments are part of a formal document, which gives the context in which to evaluate the formal content. In our use case, there is no formal context: the informal and the formal documents are strictly separated, so the formal text has to exist already, and is only referred to from a natural-language document.

We could provide an extension that allows the author to specify the formal context in which formal text is evaluated. This would allow her to write new examples based on an existing formalization, or combine literate and non-literate approaches. This is an appealing idea, but beyond the scope of this paper.

Feedback. In our use case, the natural language text only refers to the formal text, and does not feed back any formal content into the formal document. In Isar, it is possible to prove new lemmas in an antiquotation, but Wenzel notes in his thesis [28, page 65] that antiquotations printing well-typed terms, propositions and theorems are the most important ones in practice.

With these considerations in mind, we adopt the following syntax, based on the antiquotations: `@{ type reference [options] }`. The main element is `reference`, which is either a path in the Wiki or an external URL, pointing to a formal entity of the given `type`. We will discuss possible types in the next section.

The `options` element instructs the renderer of the Wiki about how to render the included entity. Compared to Isar, it has swapped positions with `reference` because it provides rendering settings, and no instructions to a formal tool. This means that they are processed last, after the reference has been processed to an object. Possible uses include flagging whether or not to include the proof of a theorem, or the level of detail that should be shown when including a snippet.

The `reference` points at an annotated object, by giving the location of the document it occurs in and the name given in the annotation for that object. The `type` corresponds to the type in the annotation of the object: it serves as a

disambiguation mechanism, but can be enforced in a more strict manner. If the system cannot find a reference of the given type, it should fail in a user friendly way: in our implementation, we inline reference in the output, marked up to show it is not found. Inspired by MediaWiki, we color it red, and put a question mark after it. An addition to this would be to make this rendering a link, through which the author can write the formal reference, or search for similar objects.

The antiquotation for the Coq code in Figure 11 is `@{oo:Definition CoqBinomialCoefficient#C}`. It points to the Definition C, found in the location (a Wiki page) `CoqBinomialCoefficient`. This reference gets resolved into the HTML shown in the screenshot.

4 Annotation of Types and Content

For transforming antiquotations to HTML, we could implement ad hoc reference resolution mechanisms specific to particular formal systems. Then, any new formal system would require building another specific dereferencing implementation from scratch. We present a more scalable approach with lower requirements for formal systems. We enrich the HTML export of the formal texts with annotations, which clearly mark the elements that authors can refer to. The Wiki can resolve them in a uniform way: when an author writes an antiquotation, the system can dereference it to the annotated HTML, without further requirements on the structure of the underlying formal texts.

This section introduces the two main kinds of annotations that are relevant here; the next section explains how to put them into formal texts. We are interested in annotating an item of formalized mathematics with its mathematical *type* (such as definition, theorem, proof), and annotating it by pointing to related *content* (such as pointing from a formalized proof to the Wikipedia article that gives an informal account of the same proof). Type annotation requires a suitable annotation *vocabulary*, whereas we had to identify suitable *datasets* as targets for content annotation.

4.1 The Type Vocabulary of the OMDoc Ontology

The OMDoc ontology provides a wide supply of types of mathematical knowledge items, as well as types of *relations* between them, e.g. that a proof proves a theorem [16,17]. It is a reimplementaion of the conceptual model of the OMDoc XML markup language [15] for the purpose of providing semantic Web applications with a vocabulary of structures of mathematical knowledge.⁸ It is thus one possible vocabulary (see [17] for others) applicable to the lightweight annotation of mathematical resources on the Web desired here, without the need to translate them from their original representation to OMDoc XML.

The OMDoc language has originally been designed for exchanging formalizations across systems for, e.g., structured specification, automated verification,

⁸ We use the terms “ontology” and “vocabulary” synonymously.

and interactive theorem proving [15]. OMDoc covers a large subset of the concepts of common languages for formalized mathematics, such as Mizar or Coq; in fact, partial translations of the latter languages to OMDoc have been implemented (see, e.g., [5]).

The OMDoc *ontology* covers most of the concepts that the OMDoc language provides for mathematical statements, structured proofs, and theories. Item types include *Theory*, *Symbol* [Declaration], *Definition*, *Assertion* (having subtypes such as *Theorem* or *Lemma*), and *Proof*; types of relations between such items include *Theory–homeTheoryOf–<any type of statement>*, *Symbol–hasDefinition–Definition*, and *Proof–proves–Theorem*. The ontology leaves the representation of document structures without a mathematical semantics, such as sections within a theory that have not explicitly been formalized as subtheories, to dedicated document ontologies (cf. [17]).

4.2 Datasets for Content Annotation

Our main use case for content annotation is annotating formalizations with related informal representations, but added-value services may still benefit from the latter having a *partial* formal semantics. Consider linking a formalized proof to a Wikipedia article that explains a sketch and the historical or application context of the proof.⁹ The information in the Wikipedia article (such as the year in which the proof was published) is not immediately comprehensible to Web services or search engines. For this purpose, DBpedia¹⁰ makes the contents of Wikipedia available as a linked open dataset.¹¹

Further suitable targets for content annotation of mathematical formalizations – albeit not yet available as machine-comprehensible linked open data – include the PlanetMath encyclopedia, the similar ProofWiki, and Wolfram’s MathWorld.¹²

In the interest of machine-comprehensibility, the links from the annotated sources to the target dataset should be *typed*. The two most widely used link types, which are also widely supported by linked data clients, are *rdfs:seeAlso* (a generic catch-all, which linked data clients usually follow to gather more information) and *owl:sameAs* (asserting that all properties asserted about the source also hold for the target, and vice versa). The OMDoc ontology furthermore provides the link type *formalizes* for linking from a formalized knowledge item to an informal item that verbalizes the former, and the inverse type *verbalizes*.

⁹ The Wikipedia category “Article proofs” lists such articles; see http://en.wikipedia.org/wiki/Category:Article_proofs.

¹⁰ <http://dbpedia.org>

¹¹ A collection of RDF descriptions accessible by dereferencing their identifiers [11].

¹² See <http://www.planetmath.org>, <http://www.proofwiki.org>, and <http://mathworld.wolfram.com>, respectively.

5 Annotating Formal Texts

Now that we have established *what* to annotate formal texts with, we need to look at the *how*. Considering that the formal documents are stored on the Web, we assume that each document has an HTML representation. Indeed, the systems we support in our prototype each have some way of generating appropriate type annotations.

Text parts are annotated by enclosing them into HTML elements that carry the annotations as RDFa annotations. RDFa is a set of attributes that allows for embedding RDF graphs into XML or HTML [2]. For identifying the annotated resources by URIs, as required by RDF, we reuse the identifiers of the original formalization.

Desired Results. Regardless of the exact details of the formal systems involved, and their output, the annotation process generally yields HTML+RDFa, which uses the OMDoc ontology (cf. Section 4.1) as a vocabulary. For example, if the formal document contains an HTML rendition of the Binomial Theorem, we expect the following result (where the prefix *oo:* has been bound to the URI of the OMDoc ontology¹³):

```
<span typeof="oo:Theorem" about="#BinomialTheorem">...</span>
<span typeof="oo:Proof"><span rel="oo:proves" resource="#BinomialTheorem"/>
...</span>
```

The “...” in this listing represent the original HTML rendition of the formal text, possibly including the information that was used to infer the annotations now captured by the RDFa attributes. @about assigns a URI to the annotated resource; here, we use fragment identifiers within the HTML document.

In this example, we wrap the existing HTML in span elements, because in most cases, this preserves the original rendering of the source text. In particular, empty spans, as typically used when there is no other HTML element around that could reasonably carry some RDFa annotation, are invisible in the browser. If the HTML of the source text contains div elements, it becomes necessary to wrap the fragment in a div instead of a span.

Mizar Texts. Mizar processing consists of several passes, similar in spirit to those used in compilation of languages like Pascal and C. The communication between the main three passes (parsing, semantic analysis, and proof checking) is likewise file-based. Since 2004, Mizar has been using XML as its native format for storing the result of the semantic analysis [23]. This XML form has been since used for producing disambiguated (linked) HTML presentation of Mizar texts, translating Mizar texts to ATP formats, and as an input for a number of other systems. The use of the XML as a native format guarantees that it remains up-to-date and usable for such external uses, which has been an issue with a number of ad-hoc ITP translations created for external use.

¹³ <http://omdoc.org/ontology#>

This encoding has been gradually enriched to contain important presentational information (e.g., the original names of variables, the original syntax of formulas before normalization, etc.), and also to contain additional information that is useful for understanding of the Mizar texts, and ATP and Wiki functions [25,24] over them (e.g., showing the thesis computed by the system after each natural deduction step, linking to ATP calls/explanations, and section editing in a Wiki).

We implemented the RDF annotation of Mizar articles as a part of the XSL transformation that creates HTML from the Mizar semantic XML format. While the OMDoc ontology defines vocabulary that seems suitable also for many Mizar internal proof steps, the current Mizar implementation only annotates the main top-level Mizar items, together with the top-level proofs. Even with this limitation this has already resulted in about 160000 annotations exported from the whole MML¹⁴ which is more than enough for testing the Agora system. The existing Mizar HTML namespace was re-used for the names of the exported items, such that, for example, the Brouwer Fixed Point Theorem¹⁵

```

:: $N Brouwer Fixed Point Theorem
theorem Th14:
  for r being non negative (real number), o being Point of TOP-REAL 2,
    f being continuous Function of Tdisk(o,r), Tdisk(o,r)
  holds f has_a_fixpoint
proof ...

```

gets annotated as¹⁶

```

<div about="#T14" typeof="oo:Theorem">
  <span rel="owl:sameAs"
    resource="http://dbpedia.org/resource/Brouwer_Fixed_Point_Theorem"/> ...
  <div about="#PF23" typeof="oo:Proof"><span rel="oo:proves" resource="#T14"/> ... </div>
</div>

```

Apart from the appropriate annotations of the theorem and its proof, an additional *owl:sameAs* link is produced to the DBpedia (Wikipedia) “Brouwer_Fixed_Point_Theorem” resource. Such links are produced for all Mizar theorems and concepts for which the author defined a long (typically well-known) name using the Mizar `::$N` pragma. Such pragmas provide a way for the users to link the formalizations to Wikipedia (DBpedia, ProofWiki, PlanetMath, etc.), and the links allow the data consumers (like Agora) to automatically mesh together different (Mizar, Coq, etc.) formalizations using DBpedia as the common namespace.

Coq Texts. Coq has access to type information when verifying a document. This information is written into a *globalization* file, which lists types and cross-references on a line/character-offset basis. Coq’s HTML renderer, Coqdoc, processes this information to generate hyperlinks between pages, and style parts of

¹⁴ MML version 4.178.1142 was used, see

http://mizar.cs.ualberta.ca/~mptp/7.12.02_4.178.1142/html/

¹⁵ http://mizar.cs.ualberta.ca/~mptp/7.12.02_4.178.1142/html/brouwer.html#T14

¹⁶ T14 is a *unique internal* Mizar identifier denoting the theorem. Th14 is a (possibly non-unique) *user-level* identifier (e.g., Brouwer or SK300 would result in T14 too).

the document according to the given types. Coqdoc is implemented as a single-pass scanner and lexer, which reads a Coq proof script and outputs HTML (or \LaTeX) as part of the lexing process.

The resulting HTML page contains the information we defined in Section 4, but serves this in an unstructured way: individual elements of the text get wrapped in `span` elements corresponding to their syntactical class, and there is no further grouping of this sequence of `spans` in a more logical entity (e.g. a `<div id="poly_id" class="lemma">...</div>`), which would be addressable from our syntax. In particular, it puts the name anchor around a theorem's label, instead of around the entire group.

For example, the following Coq code:

```
Lemma poly_id: forall a, a -> a.
```

gets translated into the following HTML fragment (truncated for brevity and whitespace added for legibility):

```
<span class="id" type="keyword">Lemma</span>
<a name="poly_id"><span class="id" type="lemma">poly_id</span>
</a>...
```

Aside from the fact that the HTML is not valid (`span` elements do not allow `@type` attributes), it has the main ingredients we are interested in extracting for annotation (type and name), but no indication that the keyword `Lemma`, the identifier `poly_id` following it, and the statement `forall a, a -> a.` are related. The problem worsens for proofs: blocks of commands are not indicated as a proof, and there is no explicit relation between a statement and its proof, except for the fact that a proof always directly follows a statement. This makes the Coqdoc-generated HTML not directly suitable for our purpose; we need three steps of post-processing:

1. **Group Objects:** The first step we take is grouping the ‘forest’ of markup elements that constitutes a command for Coq in a single element. This means parsing the text within the markup, and gathering the elements containing a full command in a new element.
2. **Export Type Information:** We then export the type information from Coq to the new element. We extract this information from the `@type`, derive the corresponding OMDoc type from it, and put that into an RDFa `@typeof` attribute.
3. **Explicit Subject Identification:** The final step is extracting `@name` and putting it into `@about`, thus reusing it as a subject URI.

After post-processing, we obtain the desired annotated tree, containing the HTML generated by Coqdoc.

The approach introduced here does not yet allow us to indicate the proof blocks, for which we do need to modify Coqdoc. The adaption is fairly straightforward: each time the tool notices a keyword starting a proof, it outputs the start of a new span, ``. Similarly, the adapted tool outputs `` when encountering a keyword signaling the end of a proof.

Isar Texts. For Isar texts, the annotation process is still in development. We make use of the Isabelle/Scala [27] library to generate HTML pages based on the proof structure, already containing the annotation of a page. Because the process has access to the full proof structure, it is easy to generate annotations: the main obstacle is that the information about the identifiers at this level does not distinguish between declaration and use, so it is difficult to know what items to annotate with an @about.

6 System: Agora

We have implemented the mechanisms described in this paper as part of the Agora prototype.^[17] A current snapshot of the source can be found in our code repository.^[18] Agora provides the following functionality, grouped by the tasks in the main use case. Writing and rendering the narrative is illustrated by the Agora page about the binomial coefficient.^[19]

Formalization. Agora allows the author to write her own formalizations grouped in *projects*, which resemble repositories of formal and informal documents. Agora has some support for verifying Coq formalizations, with a rudimentary editor for changing the files. Alternatively, it allows an author to synchronize her working directory with the system (currently, write access to the server is required for this). Proof scripts from this directory are picked up, and provided as documents. Agora also scrapes Mizar’s MML for HTML pages representing theories, and includes them in a separate project. Regardless of origin and editing methods, the proof scripts are rendered as HTML, and annotated using the vocabulary specified in Section 4.

Narratives. To allow the author to write natural language narratives, we provide the Creole Wiki syntax [21], which allows an author to use a lightweight markup syntax. Next to this markup and the antiquotation described next, the author can write formulae in L^AT_EX syntax, supported by the MathJax^[20] library.

Antiquotation. The author can include formal content from any annotated page by using the antiquotation syntax, just giving page names to refer to pages within Agora, or referring to other projects or URLs by writing a reference of the form: @{type location#name}. For example, the formalization of the binomial coefficient in Coq is included in the Wiki, so it can be referred to by @{oo:Definition CoqBinomialCoefficient#C}. On the other hand, the Mizar definition is given at an external Web page. Because the URL is rather long, the antiquotation is @{oo:Definition mml:binom.html#D22}. In this reference, mml is

¹⁷ <http://mws.cs.ru.nl/agora>

¹⁸ <https://bitbucket.org/Carst/agora>

¹⁹ http://mws.cs.ru.nl/agora/cicm_sandbox/BinomialCoefficient

²⁰ <http://www.mathjax.org>

a prefix, defined using the Agora-specific command `@{prefix mml=http://mizar.cs.ualberta.ca/~mptp/7.12.02_4.178.1142/html}`. We do not consider prefixes a part of the “core” syntax, as another implementation could restrict the system to only work within a single Wiki, causing the links to be (reasonably) short.

Rendering a page written in this way, Agora transforms the Wiki syntax into HTML using a modified Creole parser. The modification takes the antiquotations and produces a placeholder `div` containing the type, reference and repository of the antiquotation as attributes. When the page is loaded, the placeholders are replaced asynchronously, by the referred-to entities. This step is necessary to prevent very long loading times on pages referring to many external pages. When the content is included, it is pre-processed to rewrite relative links to become absolute (with the source page used as the base URL), a matter of a simple library call.

An alternative to asynchronously fetching the referenced elements would be to cache them when the page is written. This approach could be combined with the asynchronous approach implemented, and would allow authors to refer to content that would, inevitably, disappear. We currently have not implemented it, because it requires some consideration in the scope of Agora’s storage model.

Appearance of Included Content. The appearance of included snippets depends on several things:

- Cascading Style Sheets (CSS) are used to apply styling to objects that have certain attributes. When including the snippets, they can either be styled using the information in the source document (because the syntax is marked up according to rules for a specific system), or the styling can be specified in the including document (to make the rendered document look more uniform). In the implementation, we statically include the CSS files from the source text. This is manageable due to the small number of included systems, but requires further consideration.
- The system could use the included snippets to present the data using an alternative notation than the plain text that is typical for interactive theorem provers. This approach would require some system-dependent analysis of the included snippets, maybe going further than just HTML inclusion. The gathered data could then be used to render the included format in a new way, either specified by the author of the source text, or the author of the including text. This approach of “re-rendering” structured data was taken as part of the MoWGLI project [3], where the author of an informal text writes it as a *view* on a formal structure, including transformations from the formal text to (mathematical) notation.

Because our approach intends to include a wide variety of HTML-based documents, we do not consider this notational transformation viable in general: it requires specific semantic information provided by the interactive theorem prover, which is not preserved in the annotation process described in Section 5, possibly not even exposed by the prover. However, where we have this information available, it would be good to use it to make a better looking rendered result.

7 Conclusions and Future Work

This paper describes a mechanism for documenting formal proofs in an informal narrative. The narrative includes *pointers* to objects found in libraries of formalized mathematics, which have been *annotated* with appropriate types and names. The mechanism has been implemented as part of the Agora prototype. Our approach is Web-scalable in that the Agora system is independent from a particular formalized library: It may be installed in a different place, it references formal texts by URL, and it does not make any assumptions about the system underlying the library, except requiring an HTML+RDFa export. As future work, we see several opportunities for making the mechanisms more user friendly:

Include More Systems. By including more systems, we increase the number of objects an author can refer to when writing a Wiki page. Because we made our annotation framework generic, this should not be a very difficult task, and single documents could be annotated by authors on the fly, if necessary.

Provide Other Methods for Reference. At the moment, the reference part of our antiquotations is straightforward: the author should give a page and the identifier of the object in this page. It would be interesting to allow the author to use an (existing) query language to describe what item she is looking for, and use this to find objects in the annotated documents.

Improve Editing Facilities. Agora currently has a simple text box for editing its informal documents. We could provide some feedback to the author by showing a preview of the marked up text, including resolved antiquotations. More elaborately, we could provide an ‘auto-completion’ option, which shows the possible objects an author can refer to, limited by type and the partial path: if the author writes `@{oo:Theorem Foo#A}`, the system provides an auto-completion box showing all the theorems in the “Foo” namespace, starting with “A”. This lookup could be realized in a generic way, abstracting from the different formalizations, by harvesting the RDFa annotations into an RDF database (“triple store”) and implementing a query in SPARQL.

Consistency. The current design of the mechanisms already provides a better robustness than just including objects by giving a location, but can still be improved to deal with objects changing names. A solution would be to give objects an unchanging identifier and a human-readable name, and storing the antiquotation as a reference to this identifier. When an author edits a document containing an antiquotation, the name is looked up, and returned in the editable text.

Despite these shortcomings, we believe we have made significant steps towards a system in which authors can document formal mathematics by pointing and writing, without having to commit prematurely to a specific workflow, such as literate proving, or even a tool chain, because representations of (formalized) mathematics can be annotated after they have been generated.

References

1. Resource Description Framework (RDF): Concepts and abstract syntax. Recommendation, W3C (2004), <http://www.w3.org/TR/rdf-concepts>
2. RDFa in XHTML: Syntax and processing. Recommendation, W3C (October 2008), <http://www.w3.org/TR/rdfa-syntax>
3. Asperti, A., Geuvers, H., Loeb, I., Mamane, L.E., Coen, C.S.: An interactive algebra course with formalised proofs and definitions. In: Kohlhase [14], pp. 315–329
4. Autexier, S., Calmet, J., Delahaye, D., Ion, P.D.F., Rideau, L., Rioboo, R., Sexton, A.P. (eds.): AISC 2010. LNCS, vol. 6167. Springer, Heidelberg (2010)
5. Bancerek, G., Kohlhase, M.: Towards a Mizar Mathematical Library in OMDoc format. In: Matuszewski, R., Zalewska, A. (eds.) From Insight to Proof: Festschrift in Honour of Andrzej Trybulec. Studies in Logic, Grammar and Rhetoric, vol. 10(23), pp. 265–275. University of Białystok (2007)
6. Bancerek, G., Rudnicki, P.: A compendium of continuous lattices in MIZAR. *J. Autom. Reasoning* 29(3-4), 189–224 (2002)
7. Blackwell, A.F., Green, T.R.G.: Cognitive dimensions of information artefacts: a tutorial. Tutorial (1998), <http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDtutorial.pdf>
8. Cairns, P., Gow, J.: Literate proving: Presenting and documenting formal proofs. In: Kohlhase (ed.) [14], pp. 159–173
9. The Coq wiki, Browsable online at <http://coq.inria.fr/cocorico>
10. The Coq mailing list, coq-club@inria.fr
11. Heath, T., Bizer, C.: Linked Data: Evolving the Web into a Global Data Space. Morgan & Claypool (2011)
12. Heinz, C., Moses, B.: The listings package. Technical report, CTAN (2007), <http://www.ctan.org/tex-archive/macros/latex/contrib/listings>
13. The Isabelle mailing list, cl-isabelle-users@lists.cam.ac.uk
14. Kohlhase, M. (ed.): MKM 2005. LNCS (LNAI), vol. 3863. Springer, Heidelberg (2006)
15. Kohlhase, M.: OMDoc – An Open Markup Format for Mathematical Documents [version 1.2]. LNCS (LNAI), vol. 4180. Springer, Heidelberg (2006)
16. Lange, C.: OMDoc ontology (2011), <http://kwarc.info/projects/doc0nto/omdoc.html>
17. Lange, C.: Ontologies and languages for representing mathematical knowledge on the semantic web. *Semantic Web Journal* (in press, 2012)
18. Lange, C., Urban, J. (eds.): Proceedings of the ITP 2011 Workshop on Mathematical Wikis (MathWikis). CEUR-WS, vol. 767 (2011)
19. The Mizar mailing list, mizar-forum@mizar.uwb.edu.pl
20. The Mizar wiki, Browsable online at <http://wiki.mizar.org>
21. Sauer, C., Smith, C., Benz, T.: Wikicreole: a common wiki markup. In: WikiSym 2007, pp. 131–142. ACM, New York (2007)
22. The homotopy type theory blog, <http://homotopytypetheory.org/>
23. Urban, J.: XML-izing Mizar: making semantic processing and presentation of MML easy. In: Kohlhase (ed.) [14], pp. 346–360
24. Urban, J., Alama, J., Rudnicki, P., Geuvers, H.: A wiki for Mizar: Motivation, considerations, and initial prototype. In: Autexier, et al. (eds.) [4], pp. 455–469
25. Urban, J., Sutcliffe, G.: Automated reasoning and presentation support for formalizing mathematics in Mizar. In: Autexier, et al. (eds.) [4], pp. 132–146

26. van Eekelen, M.C.J.D., Geuvers, H., Schmaltz, J., Wiedijk, F.: ITP 2011. LNCS, vol. 6898. Springer, Heidelberg (2011)
27. Wenzel, M.: Isabelle as Document-Oriented Proof Assistant. In: Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.) *Calculemus/MKM 2011*. LNCS (LNAI), vol. 6824, pp. 244–259. Springer, Heidelberg (2011)
28. Wenzel, M.M.: *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Technische Universität München (2002)

An Essence of SSReflect

Iain Whiteside, David Aspinall, and Gudmund Grov

CISA, School of Informatics
University of Edinburgh
Edinburgh EH8 9AB, Scotland

Abstract. SSReflect is a powerful language for proving theorems in the Coq system. It has been used for some of the largest proofs in formal mathematics thus far. However, although it constructs proofs in a formal system, like most other proof languages the semantics is informal making it difficult to reason about such proof scripts. We give a semantics to a subset of the language, using a hierarchical notion of proof tree, and show some simple transformations on proofs that preserve the semantics.

1 Introduction

The SSReflect language for Coq was initially developed by Gonthier to facilitate his proof style of *small scale reflection* during the pioneering formalisation of the Four Colour Theorem (FCT), [7,9,14]. SSReflect provides powerful matching facilities for precise rewriting, so-called *views* offering the power of reflection, and a host of language constructs for managing the large numbers of variables and assumptions typical of the combinatorial proofs encountered in the FCT. The language has matured and is now being used more widely in the Coq community [15,13]. In particular, Gonthier and his team are using it in their formalisation of finite group theory and the Feit-Thompson theorem [8].

As in any programming or proof language, it is all too easy to write poorly structured – even unreadable – scripts during exploration and it is a tedious, often error-prone task to *refactor* proofs to make them presentable, to have good *style* [12]. Indeed, Gonthier claims to have spent months refactoring his proof of the FCT. The notion of “good” and “bad” style is cultural, and one reason for focussing on SSReflect is the clear ideas about what this style should be. Indeed, a large part of the language – the part that we are most keenly interested in – facilitates a style of proof that is designed to create proof scripts that are *robust*, *maintainable*, and *replayable*. Such proofs involves ensuring each line (or *sentence*) has a clear meaning mathematically (related formula manipulations, an inductive step, etc), keeping scripts as linear as possible, emphasising important goals, as well as supporting many convenient naming conventions.

The challenge is to provide refactorings that can be automated and, crucially, can be proved *correct*. That is, performing the refactoring will not break a proof. Unfortunately many existing proof languages, including SSReflect, have no formal semantics and their behaviour is determined by execution on goals, making

it difficult to relate equivalent proof scripts. In previous work [19], we investigated proof refactorings – as a structured and automatable way to transform a “bad” script into a “good” script – on a formally defined Isar-style declarative proof language [18]. Here we make a step in this direction for SSReflect by first modelling the semantics of (a subset of) the language, which we call *eSSence*, then demonstrating how some simple transformations may be rigorously defined. Our semantics is based on a tactic language called Hitac [1], which constructs hierarchical proof trees (or Hiproofs) when executed. Hiproofs [4] offer the advantage that one can view the proof tree at many levels of abstraction: hiding or showing as much detail as desired. We use Hiproofs as an underlying proof framework because the structured proofs written in SSReflect have a very natural hierarchical interpretation and give us novel ways to view the proof.

Contributions. We have identified the main contributions of our work as 1) providing a clear operational semantics for the eSSence language; and, 2) providing some initial refactorings of SSReflect proofs. Furthermore, we believe this presentation will help disseminate both the novel features of the language – which are by no means Coq-specific – and the Hiproof and Hitac formalisms.

Outline. In the next section, we introduce the eSSence language by example and briefly sketch the Hitac and Hiproof formalisms. Section 3 introduces a simple type theory as the underlying logic for our language. The syntax and semantics of eSSence is described in Sections 4 and 5. We then show how to use the semantics to refactor our example in Section 6, before concluding in Section 7.

2 Background

The simple eSSence (and SSReflect) script, shown below left, is a proof of $(A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow C$.

```

move => h1 h2 h3.
move : h1.
apply.
-by [].
-apply: h2.
  by [].
move => AiBiC AiB Atrue.
apply: AiBiC ; first by [].
by apply: AiB.
```

An eSSence proof is a *paragraph* that, in this case, consists of three *sentences* and then two nested paragraphs (the first a single sentence long, the second two sentences). The hyphens marking the start of each paragraph are *annotations* to explicitly show where the proof has branched. Each sentence operates on the first goal in a stack and any subgoals resulting from the execution of the sentence are pushed back on the top of the stack. We explain each sentence:

1. The `move` tactic here plays an explanatory role (acting as the identity tactic). The work is done by ‘:’ and ‘=>’, which are actually tacticals to *pop* and *push*

variables and assumptions to/from the context. This first sentence extends the context to $h1 : (A \rightarrow B \rightarrow C)$, $h2 : (A \rightarrow B)$, and $h3 : A$ with the goal being simply C .

2. The sentence `move : h1.` will then *push* $h1$ back into the goal, transforming it to $(A \rightarrow B \rightarrow C) \rightarrow C$. Note that the context is unordered, we can push arbitrary context elements as long as the context stays well-formed.
3. In the first branch, `apply` attempts to match the conclusion of the goal with the conclusion of the first assumption – motivating the previous step – then breaks down the assumption as new subgoals - in this case, two: A and B .
4. The `by` tactical attempts to solve a goal with the supplied tactic (alongside default automation applied after). The `[]` means that the goal is trivial and is solved by the default automation – the assumption $h3$ is used to solve A .
5. The second branch requires further application using $h2$ in the first sentence and solving by assumption in the second. In this branch, however, `apply:`, behaves differently from `apply` as it takes arguments, the term to be applied. With a single argument, for example $h2$, this behaves as `move : h2.` followed by `apply`.

In this example, we see two main features of the language: clustering all book-keeping operations into two tacticals (`:` and `=>`) and providing structure and robustness to scripts using indentation, annotation, and `by`. The language has many more constructs, but this gives a good flavour.

However, this isn't a very well-presented script. It is too verbose and the assumption names do not convey any information. We can apply structured changes to improve the proof. We can compress `move : h1.` and `apply.` into a single line, using the `THEN` tactical (`;`) `move: h1 ; apply.` Furthermore, we can transform it to simply `apply: h1` as `move` behaves as an identity tactic. Finally, we can merge the *obvious* steps into the same line and rename the hypotheses. To refactor `apply.` and `-by [].`, we utilise the `first` tactical whose supplied tactic operates only on the first subgoal of a branch. The resulting script is displayed above right. Now, each sentence can be understood as an – albeit simple – mathematical step. In Section 6, we show how to give provably correct transformations that can achieve this refactoring.

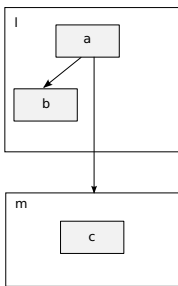


Fig. 1. A Hiproof

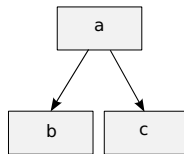


Fig. 2. The skeleton

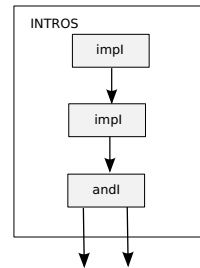


Fig. 3. INTROS

Hierarchical proof. As an underlying language for eSSence proofs, we use *Hiproofs*. First investigated by Denney et al in [4], Hiproofs are a hierarchical representation of the proof trees constructed by tactics. The hierarchy makes explicit the relationship between tactic calls and the proof tree constructed by these tactics. An abstract example of a Hiproof is given in Figure 1, where a , b , and c are called *atomic tactics* i.e. the inference rules of the language. Figure 1 reads as follows: the abstract tactic l first applies an atomic tactic a . The tactic a produces two subgoals; the first is solved by the atomic tactic b within the application of l . Thus, the high-level view is tactic l produces a single subgoal, which is then solved by the tactic m . The underlying proof tree, called the *skeleton*, is shown in Figure 2. More concretely, Figure 3 shows the application of an *INTROS* tactic as a Hiproof. Following [1], we give a syntactic description of Hiproofs:

$$s ::= a \mid id \mid swap \mid [l]s \mid s ; s \mid s \otimes s \mid \langle \rangle$$

Sequencing ($s ; s$) corresponds to composing boxes by arrows, tensor ($s \otimes s$) places boxes side-by-side, and labelling ($[l]s$) introduces a new labelled box. Identity (id) and empty ($\langle \rangle$) are units for $;$ and \otimes respectively. A *swap* switches the order of two goals. Labelling binds weakest, then sequencing with tensor binding most tightly. We can now give a syntactic description of the Hiproof in Figure 1: ($[l]a ; b \otimes id$); $[m] c$. Atomic tactics, a , come from a fixed set \mathcal{A} and what we call an *atomic tactic* is an inference rule schema:

$$\frac{\gamma_1 \cdots \gamma_n}{\gamma} a \in \mathcal{A}$$

stating that the atomic tactic a , when applied to the goal γ , breaks it down into subgoals $\gamma_1, \dots, \gamma_n$. We say Hiproofs are defined in terms of a *derivation system*, which instantiates the *atomic goals* $\gamma \in \mathcal{G}$ and *atomic tactics* $a \in \mathcal{A}$. A Hiproof is *valid* if it is well-formed and if atomic tactics are applied correctly. Validation is defined as a relation on lists of goals $s \vdash g_1 \rightarrow g_2$, and is the key proof checking property. We expand on this notion of derivation system and provide an instantiation for eSSence in Section 3.

Hierarchical tactics. The Hitac language extends Hiproofs as follows:

$$t ::= \dots \mid assert \gamma \mid t \mid t \mid name(t, \dots, t) \mid X$$

In addition to the standard Hiproof constructs, goal assertions ($assert \gamma$) can control the flow; alternation ($t \mid t$) allows choice; and, defined tactics ($name(t, \dots, t)$) and variables (X) allow us to build recursive tactic programs. Tactic evaluation is defined relative to a *proof environment*, specifying the defined tactics available. Evaluation of a tactic is defined as a relation

$$\langle g, t \rangle \Downarrow_{\mathcal{E}}^t \langle s, g' \rangle,$$

which should be read as: ‘the tactic t applied to the list of goals g returns a Hiproof s and remaining subgoals g' , under the proof environment \mathcal{E} ’.

To illustrate, we give the evaluation rules for *sequencing*, *tensor*, and *defined tactics* below, where we write \overline{X}_n as shorthand for the variable list $[X_1, \dots, X_n]$. For a full presentation see [11].

$$\frac{\langle g_1, t_1 \rangle \Downarrow_{\mathcal{E}}^t \langle s_1, g \rangle \quad \langle g, t_2 \rangle \Downarrow_{\mathcal{E}}^t \langle s_2, g_2 \rangle}{\langle g_1, t_1 ; t_2 \rangle \Downarrow_{\mathcal{E}}^t \langle s_1 ; s_2, g_2 \rangle} \quad (\text{B-TAC-SEQ})$$

$$\frac{\langle g_1, t_1 \rangle \Downarrow_{\mathcal{E}}^t \langle s_1, g'_1 \rangle \quad \langle g_2, t_2 \rangle \Downarrow_{\mathcal{E}}^t \langle s_2, g'_2 \rangle}{\langle g_1 @ g_2, t_1 \otimes t_2 \rangle \Downarrow_{\mathcal{E}}^t \langle s_1 \otimes s_2, g'_1 @ g'_2 \rangle} \quad (\text{B-TAC-TENS})$$

$$\frac{\mathcal{E}(\text{name}) = (\overline{X}_n, t) \quad \langle g, t[t_1/X_1, \dots, t_n/X_n] \rangle \Downarrow_{\mathcal{E}}^t \langle s, g' \rangle}{\langle g, \text{name}(t_1, \dots, t_n) \rangle \Downarrow_{\mathcal{E}}^t \langle s, g' \rangle} \quad (\text{B-TAC-DEF})$$

The evaluation rules for tactics are correct and construct valid proofs:

Theorem 1 (Correctness of Hitac semantics). *If $\langle g_1, t \rangle \Downarrow_{\mathcal{E}}^t \langle s, g_2 \rangle$ then $s \vdash g_1 \longrightarrow g_2$.*

With Hitac we can, for example, define a parameterised tactic $ALL(X)$, which applies the tactic supplied as a parameter to all subgoals, as follows:

$$\begin{aligned} ALL(X) &:= X \otimes (ALL(X) \mid \langle \rangle) \\ ID &:= ALL(id) \end{aligned}$$

The supplied tactic, X , is applied to the first subgoal and the tactic is called recursively on the remaining goals; it will succeed and apply the empty tactic when the list of goals is empty. Thus, ID can be seen as a more general identity tactic. We can also define tactics which rotate and reflect the order of subgoals:

$$\begin{aligned} NULL &:= \langle \rangle \mid id \\ ROTATE &:= [(swap \otimes ID) ; (id \otimes (ROTATE)) \mid \langle \rangle] \mid NULL \\ ROTATE_R &:= [(ID \otimes swap) ; (id \otimes (ROTATE_R)) \mid \langle \rangle] \mid NULL \\ REFLECT &:= ROTATE ; ((REFLECT \otimes id) \mid \langle \rangle) \end{aligned}$$

The difference between $ROTATE$ and $ROTATE_R$ is the direction of rotation. The tactic $ROTATE$ maps $[g_1, g_2, g_3, g_4]$ to $[g_2, g_3, g_4, g_1]$ and $ROTATE_R$ maps to $[g_4, g_1, g_2, g_3]$. These are used to define the various rotation tacticals in $SSReflect$.

3 Underlying Logic of eSSense

In keeping with the origins of $SSReflect$, we instantiate the $Hiproof$ framework with a type theory; however, we choose a less expressive theory than the Calculus of Inductive Constructions used in Coq to simplify the presentation but keep the flavour of the system. The logic is called λHOL and is well studied in e.g. Barendregt [5]. The set of types, \mathcal{T} , is given as follows:

$$\mathcal{T} ::= \mathcal{V} \mid \text{Prop} \mid \text{Type} \mid \text{Type}' \mid \mathcal{T} \mathcal{T} \mid \lambda \mathcal{V} : \mathcal{T}. \mathcal{T} \mid \Pi \mathcal{V} : \mathcal{T}. \mathcal{T}$$

where \mathcal{V} is a collection of variables. A declaration is $x : A$ where $A \in \mathcal{T}$ and $x \in \mathcal{V}$. A *context* is a finite, ordered sequence of declarations, with all subjects distinct and the set of *sorts*, $s = \{\mathbf{Prop}, \mathbf{Type}, \mathbf{Type}'\}$. Figure 4 enumerates the rules that axiomatise the notion of a typing judgement $\Gamma \vdash A : B$ (saying A has the type B in the context Γ). The pairs (s_1, s_2) are drawn from the set $\{(\mathbf{Type}, \mathbf{Type}), (\mathbf{Type}, \mathbf{Prop}), (\mathbf{Prop}, \mathbf{Prop})\}$, allowing construction of function types, universal quantification, and implication respectively.

Definition 1 (Atomic Goal). *A goal is a pair of a context, Γ and a type $P \in \mathcal{T}$, such that $\Gamma \vdash P : \mathbf{Prop}$. We will write $\Gamma \vdash P$ for goals.*

$$\begin{array}{c}
 \langle \rangle \vdash \mathbf{Prop} : \mathbf{Type} \\
 \hline
 \Gamma \vdash A : s \\
 \hline
 \Gamma, x : A \vdash x : A \\
 \\
 \Gamma \vdash A : B \quad \Gamma \vdash C : s \\
 \hline
 \Gamma, x : C \vdash A : B \\
 \\
 \Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B =_{\beta} B' \\
 \hline
 \Gamma \vdash A : B'
 \end{array}
 \qquad
 \begin{array}{c}
 \langle \rangle \vdash \mathbf{Type} : \mathbf{Type}' \\
 \\
 \Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi x : A.B) : s \\
 \hline
 \Gamma \vdash (\lambda x : A.b) : (\Pi x : A.B) \\
 \\
 \Gamma \vdash F : (\Pi x : A.B) \quad \Gamma \vdash a : A \\
 \hline
 \Gamma \vdash Fa : B[x := a] \\
 \\
 \Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2 \\
 \hline
 \Gamma \vdash (\Pi x : A.B) : s_2
 \end{array}$$

Fig. 4. The typing rules for $\lambda\mathbf{HOL}$

Atomic tactics, presented in Figure 5 as inference rules, should be read backwards: from a single goal, applying the rule gives zero or more subgoals. Side-conditions (restricting applicability) are also written above the line, but will be always the leftmost and not of a goal form.

A-INTRO(n). The *INTRO* tactic performs an introduction step. The subgoal generated is the obvious one and the assumption is given the name supplied.

A-EXACT(t). The *EXACT* tactic takes a term as a parameter and solves the goal if the term has a type convertible with the goal.

A-REFINE(t). Refinement takes a term of a convertible type to the current goal, containing proof variables – explicitly shown in the rule – and leaves the variables as subgoals. There is an additional side-condition on the rule A-REFINE: that the variables x_i cannot occur inside the $P_1 \dots P_n$.

Atomic tactics preserve *well-formedness* in the sense that given a well-formed goal (Definition 1), application of the tactic produces more well-formed goals. Another important property for a proof system is:

Conjecture 1 (Soundness of Atomic Tactics.) *The atomic tactics are sound with respect to the low-level rules of $\lambda\mathbf{HOL}$. That is, if we construct a proof term of the subgoals, then we can construct a proof term for the original goals.*

We have not proved this for our system, as it is not a key goal: rather the actual logical system serves to illustrate our approach.

$$\begin{array}{c}
\frac{\Gamma \vdash t : Q \quad P =_{\beta} Q}{\Gamma \vdash P} \quad (\text{A-EXACT}(t)) \\
\frac{(x : P) \in \Gamma \text{ for some } x}{\Gamma \vdash P} \quad (\text{A-ASSUMPTION}) \\
\frac{n \notin \Gamma \quad \Gamma, (n : T) \vdash U}{\Gamma \vdash \Pi x : T. U} \quad (\text{A-INTRO}(n)) \\
\frac{x : T \in \Gamma \quad wf(\Gamma \setminus x : T) \quad \Gamma \setminus x : T \vdash \Pi x : T. P}{\Gamma \vdash P} \quad (\text{REVERT}(x)) \\
\frac{\Gamma \vdash U : Prop \quad \Gamma \vdash U \quad \Gamma \vdash U \rightarrow P}{\Gamma \vdash P} \quad (\text{A-ASSERT}(U)) \\
\frac{t(?x_1 : P_1, \dots, ?x_n : P_n) : Q \quad P =_{\beta} Q}{\Gamma \vdash P_1 \dots \Gamma \vdash P_n} \quad (\text{A-REFINE}(t))
\end{array}$$

Fig. 5. Atomic tactics

4 The eSSence Language

The syntax for eSSence is given in Figure 6. A *sentence* is a grammar element *sstac*. The parameters *num*, *term*, and *ident* stand for numerals, terms, and identifiers respectively. The basic tactics are **move**, **apply**, and one or more rewrite steps. The **have** tactic allows forward proof and **;** is the LCF THEN and THENL tacticals. The **by** tactical ensures that the supplied tactic solves the current goal. Using the **first** and **last** tacticals, one can operate on a subset of goals and **first** *n* **last** performs subgoal rotation. Most bookkeeping operations are performed using the *discharge* and *introduction* tacticals (**:** and **=>** respectively), which pop/push assumptions and variables from/to the context. The first *iitem* in an application of the introduction tactical can be a branching pattern, allowing for the case where the corresponding tactic introduces multiple subgoals, which we can deal with simultaneously. For a full presentation and tutorial guide to the original SSReflect language, see [\[9,10\]](#).

Paragraphs. A proof script in SSReflect is simply a list of sentences, separated by full stops; however, one can optionally *annotate* a script with bullets (*, +, and -), which, along with indentation, helps make clear the subgoal flow within a script. The idea of these annotations is to use bullets to highlight where a proof branches. We build this directly into the eSSence language using the *sspara* grammar element. Paragraphs can be understood abstractly as a non-empty list of sentences followed by a possibly empty list of (indented) paragraphs. We add the restriction that in each paragraph the annotations must be the same bullets. There is no semantic difference between the various bullet symbols. The SSReflect annotation guidelines state that:

- If a tactic sentence evaluates and leaves one subgoal, then no indentation or annotation is required. If a tactic sentence introduces two subgoals – the **have** tactic, for instance – then the proof of the first goal is indented. The second is at the same level of indentation as the parent goal.

```

ssscript ::= sspara
sstac ::= dtactic
          | apply
          | apply:
          | rewrite rstep+
          | have: term [by sstac]
          | sstac ; chtac
          | by chtac
          | exact term
          | sstac ; first [num] chtacopt
          | sstac ; last [num] chtacopt
          | sstac ; first [num] last
          | sstac ; last [num] first
          | dtactic : ditem ... ditem
          | sstac => [iitemstart] iitem ... iitem
dtactic ::= move
chtac ::= sstac | [sstac | ... | sstac]
chtacopt ::= sstac | [sstac] | ... | [sstac]
iitemstart ::= iitem | [iitem* | ... | iitem*]
ssanno ::= + | - | *
sspara ::= sstac .
          |
          | sstac .
          | (ssanno sspara)*
rstep ::= ([-]term) | sitem
sitem ::= /= | // | // =
ditem ::= term
iitem ::= sitem | ipattern
ipattern ::= ipatt ident

```

Fig. 6. The eSSence language syntax

- If a sentence introduces three or more subgoals then bullets and indentation are required to mark the start of each subgoal’s proof. The last, however, is outdented to the same level as the parent.

The outdenting of the final goal is to emphasise that it is somehow more difficult or interesting; this concept motivates the tacticals for rotating subgoals. We simplify the annotations by using explicit bullets even for the case of two subgoals and also bulleting the final subgoal. This notion of structuring corresponds exactly with the hierarchy in Hiproofs - every bullet corresponds to a labelled box. Figure 7 contains examples of scripts that follow our simplified structuring guidelines (on the right is an indication of the arity of each tactic). Our semantics, given next, enforces these guidelines and in Section 6 we give a syntax and semantics for scripts without annotation and describe how to annotate a script automatically.

5 Giving Meaning to eSSence Scripts

The semantics for eSSence is based on a static translation from eSSence to Hitac, written $\llbracket \textit{sstac} \rrbracket$, giving us:

$s1. \quad [\gamma_1] \rightarrow [\gamma_2]$ $s2. \quad [\gamma_2] \rightarrow [\gamma_3]$ $s3. \quad [\gamma_3] \rightarrow []$	$s1. \quad [\gamma] \rightarrow [\gamma_1, \gamma_2, \gamma_3]$ $- s2. \quad [\gamma_1] \rightarrow []$ $- s3. \quad [\gamma_2] \rightarrow []$ $- s4. \quad [\gamma_3] \rightarrow [\gamma_4]$ $s5. \quad [\gamma_4] \rightarrow []$	$s1. \quad [\gamma] \rightarrow [\gamma_1, \gamma_2]$ $- s2. \quad [\gamma_1] \rightarrow []$ $- s3. \quad [\gamma_2] \rightarrow [\gamma_3, \gamma_4]$ $+ s4. \quad [\gamma_3] \rightarrow []$ $+ s5. \quad [\gamma_4] \rightarrow []$
---	---	---

Fig. 7. A linear script, one level of branching, and multiple branching levels

$$\frac{\langle \gamma, \llbracket sstac \rrbracket \rangle \Downarrow_{\mathcal{E}}^t \langle s, g \rangle}{\langle \gamma, sstac \rangle \Downarrow_{\mathcal{E}} \langle s, g \rangle}$$

which says that under an *environment* \mathcal{E} the tactic *sstac* applied to the goal γ results in a list of generated subgoals g and a Hiproof s if the translated Hitac tactic behaves the same. We present most of the translation rules in Figures [8](#) and [13](#). We will explain the rules in Figure [8](#) and leave the rest for the reader.

$\llbracket \text{move} \rrbracket$	$=$	$\llbracket \text{move} \rrbracket \text{ assert } (\Gamma \vdash \Pi x : A.B) \mid \text{HNF}$
$\llbracket \text{by sstac} \rrbracket$	$=$	$\llbracket \text{by} \rrbracket \llbracket sstac \rrbracket ; \text{ALL}(\text{DONE}) ; \langle \rangle$
$\llbracket sstac_1 ; \text{first sstac}_2 \rrbracket$	$=$	$\llbracket sstac_1 \rrbracket ; (\llbracket \text{first} \rrbracket \llbracket sstac_2 \rrbracket) \otimes \text{ID}$
$\llbracket sstac ; \text{first last} \rrbracket$	$=$	$\llbracket \text{FL} \rrbracket \llbracket sstac \rrbracket ; \text{REFLECT}$
$\llbracket sstac ; \text{first } k \text{ last} \rrbracket$	$=$	$\llbracket \text{FkL} \rrbracket \llbracket sstac \rrbracket ; \text{ROTATE}^k$
$\llbracket sstac \Rightarrow iitem_1 \dots iitem_n \rrbracket$	$=$	$\llbracket \Rightarrow \rrbracket \llbracket sstac \rrbracket ; \llbracket \text{ipat } iitem_1 \rrbracket ; \dots ; \llbracket \text{ipat } iitem_n \rrbracket$
$\llbracket \text{ipat ident} \rrbracket$	$=$	$\text{INTRO}(\text{id})$

Fig. 8. eSSence evaluation semantics part one

$\llbracket \text{move} \rrbracket$. The *move* tactic behaves as an identity if an introduction step is possible or transforms the goal to head normal form otherwise. We model this by providing an assertion for checking a product and applying a tactic *HNF*, which reduces the goal to head normal form. Note also that we ‘box up’ the application of *move* to abstract away from any normalisation done by this tactic. This is the first of many occasions in the semantics where we use hierarchy to hide away details.

$\llbracket \text{by sstac} \rrbracket$. This rule evaluates the primitive form of the closing tactical, applying the *DONE* tactic to all subgoals after its parameter as some sort of default automation (including *A-ASSUMPTION*). We assume it to be built

from more primitive tactics. The empty hiproof is used to fail the *by* tactical if the goals are not solved after this automation is applied. The translation is invoked recursively on *sstac*.

$\llbracket sstac_1 ; \text{first } sstac_2 \rrbracket$. In this primitive form of selection tactical, we apply *sstac*₂ to only the first subgoal generated and add hierarchy to hide the proof.

$\llbracket sstac ; \text{first last} \rrbracket$. This tactical simply reflects the subgoals and is implemented by the *Hitac* reflection tactic described earlier.

$\llbracket sstac ; \text{first } k \text{ last} \rrbracket$. To the subgoals $[g_1, g_2, g_3, g_4, g_5]$, *sstac*₁ ; **first 2 last** would result in the remaining goals looking like $[g_3, g_4, g_5, g_1, g_2]$ and it is implemented by an appropriate number of rotations. The syntax $t ;^n$ simply means a sequence of applications of a tactic t of length n i.e. $t ; \dots ; t$.

$\llbracket sstac \Rightarrow iitem_1 \dots iitem_n \rrbracket$ **and** $\llbracket ipat \text{ ident} \rrbracket$. These rules are used to translate the non-branching version of the introduction tactical, an instance of which would be *sstac* $\Rightarrow iitem_1 \dots iitem_n$. The tactic *sstac* is evaluated first; then each *iitem*_{*i*} left to right. Each *iitem* can be either a simplification item or an *ipattern* and each *ipattern* is simply an introduction step.

Paragraphs. We represent scripts abstractly as a pair of lists: a *sstac* list and a paragraph list and we extend the evaluation relation to operate on a list of goals and produce a list of subgoals and a Hiproof. The evaluation rules follow:

$$\frac{\langle \gamma, sstac \rangle \Downarrow_{\mathcal{E}} \langle s, [] \rangle}{\langle [\gamma], ([sstac], []) \rangle \Downarrow_{\mathcal{E}} \langle [S]s, [] \rangle} \quad (\text{SS-TAC})$$

$$\frac{sstacs \neq [] \quad \langle \gamma, sstac \rangle \Downarrow_{\mathcal{E}} \langle s_{\gamma}, [\gamma'] \rangle}{\langle [\gamma'], (sstacs, ssparas) \rangle \Downarrow_{\mathcal{E}} \langle s, [] \rangle} \quad (\text{SS-TACCONS})$$

$$\frac{\langle \gamma, sstac \rangle \Downarrow_{\mathcal{E}} \langle s_{\gamma}, g \rangle \quad \text{length}(g) > 1 \quad \langle g, ([], ssparas) \rangle \Downarrow_{\mathcal{E}} \langle s, [] \rangle}{\langle [\gamma], ([sstac], ssparas) \rangle \Downarrow_{\mathcal{E}} \langle ([S]s_{\gamma}), s, [] \rangle} \quad (\text{SS-PARASTART})$$

$$\frac{\langle [\gamma], sspara \rangle \Downarrow_{\mathcal{E}} \langle s_{\gamma}, [] \rangle \quad \langle g, ([], ssparas) \rangle \Downarrow_{\mathcal{E}} \langle s_g, [] \rangle}{\langle \gamma :: g, ([], sspara :: ssparas) \rangle \Downarrow_{\mathcal{E}} \langle ([P]s_{\gamma}) \otimes s_g, [] \rangle} \quad (\text{SS-PARACONS})$$

$$\langle [], ([], []) \rangle \Downarrow_{\mathcal{E}} \langle \langle \rangle, [] \rangle \quad (\text{SS-PARAEND})$$

The idea is that given a paragraph (a pair of sentence and sub-paragraph lists), we first evaluate each sentence sequentially. All sentences except the last must return one goal (SS-TACCONS). The last must either solve the goal and be the end of the paragraph (SS-TAC) or leave $n > 1$ and contain n nested paragraphs (SS-PARASTART). We apply each paragraph to each goal, then label and glue the proofs together (SS-PARACONS). Each sentence is also labeled to make clear the script structure. In this, we simply labelled each sentence and paragraph with S or P ; however, in future we plan to allow supplied names. It can be shown that the evaluation relation behaves suitably:

Proposition 1 (Correctness of Evaluation). *If $(\Gamma \vdash P) \equiv \gamma$ is a well-formed proposition, and $\langle \gamma, sstac \rangle \Downarrow_{\mathcal{E}} \langle s, g \rangle$ then s is **valid**. That is, $s \vdash [\gamma] \rightarrow g$.*

Example. Recall our proof from Section 2, shown here with sentences labelled:

```

move => AiBiC AiB Atrue.      sent1
apply : AiBiC ; first by []   sent2
by apply : AiB.                sent3
    
```

This script is parsed into a paragraph consisting of three sentences (and no additional paragraphs). At the script level, each sentence is evaluated sequentially using the rule SS-TACCONS twice and then SS-TAC to finish the proof (since it operates on a singleton sentence list and empty paragraph list). This gives us the high-level Hiproof in Figure 9. At the level of sentences, we have:

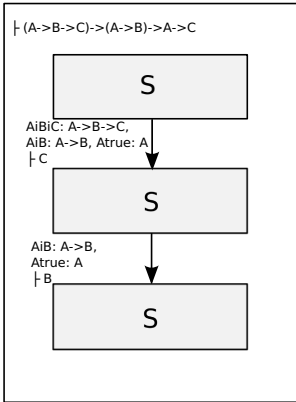


Fig. 9. High level view of Hiproof

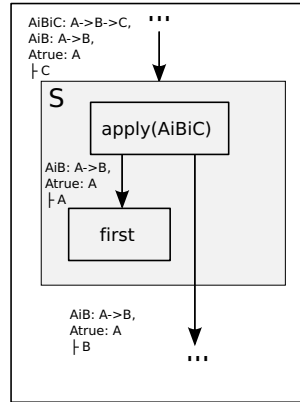


Fig. 10. The Hiproof for *sent2*

- *sent1* is translated by applying introduction tactical translation rule at the top level. During tactic evaluation, *move* behaves like an identity then each *item* applies the INTRO tactic to generate a goal:

$$AiBiC : A \rightarrow B \rightarrow C, AiB : A \rightarrow B, Atrue : A \vdash C.$$

- *sent2*, whose root is an application of the *first* tactical (where *sstac*₁ is *apply : AiBiC* and *sstac*₂ is *by []*), is then translated. The resulting tactic calls *apply* and generates *two subgoals*: *A* and *B* in a context with *AiB* and *Atrue*. The second part of the translated *first* tactical – corresponding to the closing tactical *by []* – is evaluated and solves the goal.
- The final sentence is used to solve the goal *AiB : A → B, Atrue : A ⊢ B*. and proceeds similarly to *sent2*.

Figure 10 shows one *view* of the Hiproof generated by execution of *sent2*, in context with the rest of the proof. Here we see a little of the power of Hiproofs, as we can effectively hide the details of the proof of the first subgoal.

6 Refactoring eSSence

We are now in a position to demonstrate a small number of refactorings and illustrate the general approach to showing correctness. In order to do so we first introduce a notion of *unstructured script*, which is simply a list of sentences (*sstacs*). From an eSSence script, we can easily obtain an unstructured script by simply dropping annotations, and collapsing paragraphs into a single list. We end up, for example, using the second example in Figure 7, with the script represented as $s_1 :: s_2 :: \dots s_n :: []$ (or $[s_1, \dots, s_n]$), given in Figure 11.

s1. $[\gamma] \rightarrow [\gamma_1, \gamma_2, \gamma_3]$		
s2. $[\gamma_1] \rightarrow []$	$\langle [], [] \rangle \Downarrow_{\mathcal{E}} \langle \langle \rangle, \langle \rangle \rangle$	(NS-EMP)
s3. $[\gamma_2] \rightarrow []$		
s4. $[\gamma_3] \rightarrow [\gamma_4]$	$\frac{\langle \gamma, ss \rangle \Downarrow_{\mathcal{E}} \langle s_\gamma, g_\gamma \rangle \quad n = \text{len}(g) - 1}{\langle g_\gamma @ gs, tacs \rangle \Downarrow_{\mathcal{E}} \langle s, g \rangle}$	(NS-CONS)
s5. $[\gamma_4] \rightarrow []$	$\langle \gamma :: gs, ss :: tacs \rangle \Downarrow_{\mathcal{E}} \langle (s_\gamma \otimes id^n) ; s, g \rangle$	

Fig. 11. Unstructured script

Fig. 12. Evaluation of unstructured script

We give a semantics to evaluation with Figure 12 (writing id^n for an identity tensor of length n i.e. $id \otimes \dots \otimes id$). The rule NS-CONS peels off a goal from the stack and applies the first tactic to it; it then pushes the resulting subgoals onto the stack (pre-appending to a list) and recurses with the rest of the tactics to be applied. The Hiproof is pieced together by tensoring together identity tactics to ‘skip’ the rest of the goals in the initial list, with NS-EMP dealing with the empty list case. Crucially, if an unstructured script is well-formed and evaluates successfully, we can introduce structure using an *annotation* operation. Annotation requires only information about the *arity* of each tactic (that is, the number of resulting subgoals). Annotation and flattening are dual operations and we have the following important property:

Theorem 2 (Correctness of annotation). *If we have an unstructured script, sstacs, such that: $\langle \gamma, sstacs \rangle \Downarrow_{\mathcal{E}} \langle s, g \rangle$, then: $\langle \gamma, \text{annotate}(sstacs) \rangle \Downarrow_{\mathcal{E}} \langle s', g \rangle$.*

This means, that if we wish we can ignore the structure when reasoning about refactorings as it doesn’t affect the resulting subgoals, just the Hiproof.

Transforming our example. In Section 2, we modified a simple proof script to improve its style. In particular, the specific operations we used:

- **Rename hypothesis:** to rename, for example, $h1$ to $AiBiC$.
- **Merge sentences:** to shorten the proof by collapsing to sentences into one.
- **Propagate closing tactical:** to move a **by** tactical outwards.
- **Replace move instance:** replacing it with **apply:**.

We focus only on *propagate* and *merge* in this paper.

Propagate. This refactoring moves an instance of **by** outside another tactical, if possible. There is a transformation rule for each suitable tactical, for example:

$$\frac{}{\text{propagate}(sstac_1 ; \text{by } sstac_2) \longrightarrow \text{by } sstac_1 ; sstac_2} \quad (\text{PROP-1})$$

Propagate is correct if the sentence evaluates as $\langle \gamma, sstac_1 ; \text{by } sstac_2 \rangle \Downarrow_{\mathcal{E}} \langle s, [] \rangle$ and then $\langle \gamma, \text{propagate}(sstac_1 ; \text{by } sstac_2) \rangle \Downarrow_{\mathcal{E}} \langle s', [] \rangle$, for some s' . The condition that the original sentence must successfully evaluate is called a *pre-condition* and is important as often refactorings are not universally applicable. The semantics of the *closing tactical* **by** $sstac$ relates it directly to the Hitac tactic $sstac ; ALL(DONE) ; \langle \rangle$. This means the LHS of PROP-1 is $sstac_1 ; ALL(sstac_2 ; ALL(DONE) ; \langle \rangle)$, which is equivalent to

$$(sstac_1 ; ALL(sstac_2)) ; ALL(DONE) ; \langle \rangle \equiv \text{by } sstac_1 ; sstac_2.$$

We can perform this refactoring inside a larger proof – here we only specified it on a single line – by integrating it with another refactoring called *transform sentence*, which has the obvious behaviour and is easily shown correct for unstructured scripts by a standard structural induction.

Merge sentences. There are two variations of this refactoring:

1. We merge **move** : h1. and **apply** into a single line, using the THEN tactical, since **move** : h1. generates a single subgoal.
2. Since **apply**: h1 generates two subgoals and - **by** [] is the first, use the **first** tactical to merge them as **apply**: h1 ; **first by** [].

The first version first drops annotations and works on unstructured scripts by stepping through the sentence list until the sentences to be merged are encountered. Then the following rule is applied:

$$\frac{\text{arity}(s_1) = 1}{\text{merge}(s_1, s_2, s_1 :: s_2 :: sstacs) \longrightarrow s_1 ; s_2 :: sstacs} \quad (\text{SS-MERGE-1})$$

The second case requires more sophistication. If $\text{arity}(s_1) > 1$, then we can merge it with the *first* sentence of any of the subparagraphs, using the **first** tactical. The transformation rule can be seen on structured scripts as the following, where s is the first sentence, and i is the position of the second sentence in the paragraph list. It merges the first sentence in the appropriate paragraph ($\text{head}(\text{fst}(sspara_i))$) with the supplied sentence.

$$\frac{\text{arity}(s) = n \quad sspara'_i = (\text{tl}(\text{fst}(sspara_i)), \text{snd}(sspara_i)) \quad sstac = \text{hd}(\text{fst}(sspara_i)) \quad ssparas = [sspara_1, \dots, sspara_{i-1}, sspara'_i, sspara_{i+1}, \dots, sspara_n]}{\text{merge}(s, i, ([s], [sspara_1, \dots, sspara_n])) \longrightarrow ([s ; \text{first } i [sstac]], ssparas)} \quad (\text{SS-MERGE-2})$$

Correctness of the first version is again an easy induction on unstructured scripts; however, the second case is more difficult as it involves manipulating an

$\llbracket \text{apply} \rrbracket$	$=$	$\llbracket \text{apply} \rrbracket \text{INTRO}(top)$; $(\text{REFINE}(top) \mid \text{REFINE}(top \ _)) \mid \dots$
$\llbracket \text{apply} : t_1 \dots t_n \rrbracket$	$=$	$\llbracket \text{apply} \rrbracket (\text{REFINE}(t_1(t_2 \dots t_n)) \mid \text{REFINE}(t_1 _ (t_2 \dots t_n))) \mid \dots$
$\llbracket \text{rstep rev tm} \rrbracket$	$=$	$\llbracket \text{rstep} \rrbracket \text{REWRITE}(rev, tm)$
$\llbracket \text{rewrite rstep}_1 \dots \text{rstep}_n \rrbracket$	$=$	$\llbracket \text{rewrite} \rrbracket \llbracket \text{rstep}_1 \rrbracket ; \dots ; \llbracket \text{rstep}_n \rrbracket$
$\llbracket \text{exact term} \rrbracket$	$=$	$\llbracket \text{exact} \rrbracket \text{EXACT}(tm)$
$\llbracket \text{by } \llbracket \text{sstac}_1 \rrbracket \mid \dots \mid \llbracket \text{sstac}_n \rrbracket \rrbracket$	$=$	$\llbracket \text{by} \rrbracket \llbracket \text{sstac}_1 \rrbracket ; \text{ALL}(\text{DONE}) ; \langle \rangle \mid \dots \mid$ $\llbracket \text{sstac}_n \rrbracket ; \text{ALL}(\text{DONE}) ; \langle \rangle$
$\llbracket \text{have term by sstac} \rrbracket$	$=$	$\text{ASSERT}(tm) ; (\llbracket \text{sstac} \rrbracket \otimes id)$
$\llbracket \text{sstac}_1 ; \text{sstac}_2 \rrbracket$	$=$	$\llbracket \text{sstac}_1 \rrbracket ; \text{ALL}(\llbracket \text{sstac}_2 \rrbracket)$
$\llbracket \text{sstac}_0 ; \llbracket \text{sstac}_1 \rrbracket \mid \dots \mid \llbracket \text{sstac}_n \rrbracket \rrbracket$	$=$	$\llbracket \text{sstac}_0 \rrbracket ; (\llbracket \text{sstac}_1 \rrbracket \otimes \dots \otimes \llbracket \text{sstac}_n \rrbracket)$
$\llbracket \text{sstac}_1 ; \text{last sstac}_2 \rrbracket$	$=$	$\llbracket \text{sstac}_1 \rrbracket ; ID \otimes (\llbracket \text{last} \rrbracket \llbracket \text{sstac}_2 \rrbracket)$
$\llbracket \text{sstac}_0 ; \text{last } k \llbracket \text{sstac}_1 \rrbracket \mid \dots \mid \llbracket \text{sstac}_n \rrbracket \rrbracket$	$=$	$\llbracket \text{sstac}_0 \rrbracket ; ID \otimes (\llbracket \text{last} \rrbracket \llbracket \text{sstac}_1 \rrbracket) \otimes \dots \otimes (\llbracket \text{last} \rrbracket \llbracket \text{sstac}_n \rrbracket) \otimes id_{k-1}$
$\llbracket \text{sstac}_0 ; \text{first } k \llbracket \text{sstac}_1 \rrbracket \mid \dots \mid \llbracket \text{sstac}_n \rrbracket \rrbracket$	$=$	$\llbracket \text{sstac}_0 \rrbracket ; id_{k-1} \otimes (\llbracket \text{first} \rrbracket \llbracket \text{sstac}_1 \rrbracket) \otimes \dots \otimes (\llbracket \text{first} \rrbracket \llbracket \text{sstac}_n \rrbracket) \otimes ID$
$\llbracket \text{sstac} ; \text{last first} \rrbracket$	$=$	$\llbracket LF \rrbracket \llbracket \text{sstac} \rrbracket ; \text{REFLECT}$
$\llbracket \text{sstac} ; \text{last } k \text{ first} \rrbracket$	$=$	$\llbracket LkF \rrbracket \llbracket \text{sstac} \rrbracket ; \text{ROTATE}_{_,R,k}$
$\llbracket \text{ditem term} \rrbracket$	$=$	$\text{REVERT}(tm)$
$\llbracket \text{dtactic} : \text{ditem}_1 \dots \text{ditem}_n \rrbracket$	$=$	$\llbracket \cdot \rrbracket \llbracket \text{ditem term}_n \rrbracket ; \dots ; \llbracket \text{ditem term}_1 \rrbracket ; \llbracket \text{dtactic} \rrbracket$
$\llbracket // \rrbracket$	$=$	$\text{ALL}(\llbracket \text{DONE} \rrbracket \text{DONE})$
$\llbracket // = \rrbracket$	$=$	$\text{ALL}(\llbracket \text{SIMP} \rrbracket \text{SIMP})$
$\llbracket // = \rrbracket$	$=$	$\llbracket // = \rrbracket ; \llbracket // \rrbracket$

Fig. 13. Remaining translation rules

arbitrary paragraph list. We identify the precise paragraph to refactor and use the *transform paragraph* refactoring to simplify the problem. We can then induct on the transformation over structured scripts and use equivalences in the Hitac language to show that evaluation succeeds.

7 Conclusions

In this work, we have identified and provided a semantics for an important subset of the SSReflect language, dealing primarily with proof style. Furthermore, we believe we have disseminated the principles of the language in a clearer setting and taken advantage of its inherent hierarchical nature by basing our semantics on the Hiproof framework. Importantly for our future program of work, this semantics enables us to reason about SSReflect scripts and, in particular, justify the correctness of refactorings.

Related work. Most related to our work are [17,11], where a formal semantics is given for a declarative language and a procedural language, which is used to recover proof scripts from the underlying proof term. There has also been some work on formal semantics for proof languages, such as C-zar for Coq, and the Ω mega proof language [3,2], but we are not aware of any work which specifically attempts to take advantage of this for these languages.

Refactoring is well-studied for programming languages and the literature is well surveyed by Mens et al [16]. The canonical reference for programming language refactoring is Fowler [6]. This book, widely considered to be the handbook of refactoring, consists of over 70 refactorings with a detailed description of the motivation for each refactoring and how to carry it out safely, and many of these refactorings in the domain of programming map across directly to proof.

Further work. There are many ways to progress this work. In particular, we would like to extend our semantics to cover a larger set of the SSReflect language; in particular, we would like to study *clear switches* in SSReflect, which allow the user to delete assumptions from the context. These are an important part of the language's design and make data flow explicit. In this work, we have not dealt with meta-variables. This is something that we would like to investigate, to see if our techniques could scale to a language which formally deals with meta-variables. With the refactorings described above, we have only scratched the surface of what is possible and plan to focus both on refactorings that are specific to the SSReflect language and translating refactorings that we have studied in previous work into a new language [19].

Acknowledgements. The authors would like to thank Georges Gonthier for the many useful discussions which motivated this work. The first author was supported by Microsoft Research through its PhD Scholarship Programme. The third author was supported by EPSRC grant EP/H024204/1.

References

1. Aspinall, D., Denney, E., Lüth, C.: Tactics for hierarchical proof. *Mathematics in Computer Science* 3, 309–330 (2010)
2. Autexier, S., Dietrich, D.: A Tactic Language for Declarative Proofs. In: Kaufmann, M., Paulson, L.C. (eds.) *ITP 2010*. LNCS, vol. 6172, pp. 99–114. Springer, Heidelberg (2010)
3. Corbineau, P.: A Declarative Language for the Coq Proof Assistant. In: Miculan, M., Scagnetto, I., Honsell, F. (eds.) *TYPES 2007*. LNCS, vol. 4941, pp. 69–84. Springer, Heidelberg (2008)
4. Denney, E., Power, J., Tourlas, K.: Hiproofs: A hierarchical notion of proof tree. *Electr. Notes Theor. Comput. Sci.* 155, 341–359 (2006)
5. Barendregt, H., et al.: Lambda calculi with types. In: *Handbook of Logic in Computer Science*, pp. 117–309. Oxford University Press (1992)
6. Fowler, M.: *Refactoring: improving the design of existing code*. Addison-Wesley (1999)
7. Gonthier, G.: The Four Colour Theorem: Engineering of a Formal Proof. In: Kapur, D. (ed.) *ASCM 2007*. LNCS (LNAI), vol. 5081, p. 333. Springer, Heidelberg (2008)
8. Gonthier, G., Mahboubi, A., Rideau, L., Tassi, E., Théry, L.: A Modular Formalisation of Finite Group Theory. *Rapport de recherche RR-6156*, INRIA (2007)
9. Gonthier, G., Mahboubi, A., Tassi, E.: A Small Scale Reflection Extension for the Coq system. *Rapport de recherche RR-6455*, INRIA (2008)
10. Gonthier, G., Stéphane Le, R.: An Ssreflect Tutorial. *Technical Report RT-0367*, INRIA (2009)
11. Guidi, F.: Procedural representation of cic proof terms. *J. Autom. Reason.* 44(1-2), 53–78 (2010)
12. Harrison, J.: Proof Style. In: Giménez, E. (ed.) *TYPES 1996*. LNCS, vol. 1512, pp. 154–172. Springer, Heidelberg (1998)
13. Heras, J., Poza, M., Dénès, M., Rideau, L.: Incidence Simplicial Matrices Formalized in Coq/SSReflect. In: Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.) *Calculemus/MKM 2011*. LNCS, vol. 6824, pp. 30–44. Springer, Heidelberg (2011)
14. Huet, G., Kahn, G., Paulin-Mohring, C.: *The Coq proof assistant: A tutorial* (August 2007)
15. Komendantsky, V.: Reflexive toolbox for regular expression matching: verification of functional programs in Coq+SSReflect. In: *PLPV 2012*, pp. 61–70 (2012)
16. Mens, T., Tourwe, T.: A survey of software refactoring. *IEEE Trans. Softw. Eng.* 30(2), 126–139 (2004)
17. Sacerdoti Coen, C.: Declarative representation of proof terms. *J. Autom. Reason.* 44(1-2), 25–52 (2010)
18. Wenzel, M.: Isar - A Generic Interpretative Approach to Readable Formal Proof Documents. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) *TPHOLs 1999*. LNCS, vol. 1690, pp. 167–184. Springer, Heidelberg (1999)
19. Whiteside, I., Aspinall, D., Dixon, L., Grov, G.: Towards Formal Proof Script Refactoring. In: Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.) *Calculemus/MKM 2011*. LNCS (LNAI), vol. 6824, pp. 260–275. Springer, Heidelberg (2011)

Theory Presentation Combinators^{*}

Jacques Carette and Russell O'Connor

Department of Computing and Software
McMaster University
Hamilton, Ontario, Canada
{cchette,roconn}@mcmaster.ca

Abstract. We motivate and give semantics to *theory presentation combinators* as the foundational building blocks for a scalable library of theories. The key observation is that the *category of contexts* and fibered categories are the ideal theoretical tools for this purpose.

1 Introduction

A mechanized mathematics system, to be useful, must possess a large library of mathematical knowledge, on top of sound foundations. While sound foundations contain many interesting intellectual challenges, building a large library seems a daunting task because of its sheer volume. However, as has been well-documented [5,6,13], there is a tremendous amount of redundancy in existing libraries.

Our aim is to build tools that allow library developers to take advantage of all the commonalities in mathematics so as to build a large, rich library for end-users, whilst expending much less actual development effort. In other words, we continue with our approach of developing *High Level Theories* [4] through building a network of theories, by putting our previous experiments [5] on a sound theoretical basis.

1.1 The Problem

The problem which motivates this research is fairly simple: give developers of mathematical libraries the foundational tools they need to take advantage of the inherent structure of mathematical theories, as first class mathematical objects in their own right. Figure 1 shows the type of structure we are talking about: The presentation of the theory `Semigroup` strictly contains that of the theory `Magma`, and this information should not be duplicated. A further requirement is that we need to be able to selectively hide (and reveal) this structure from end-users.

^{*} This research was supported by NSERC.

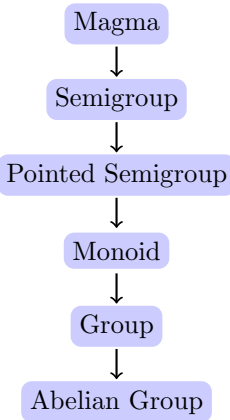


Fig. 1. Theories

The motivation for these tools should be obvious, but let us nevertheless spell it out: we simply cannot afford to spend the human resources necessary (one estimate was 140 person-years [21]; [1] explore this topic in much greater depth) to develop yet another mathematical library. In fact, as we *now* know that there is a lot of *structured* redundancy in such libraries, it would be downright foolish to not take full advantage of that. As a minor benefit, it can also help reduce errors in axiomatizations.

The motivation for being able to selectively hide or reveal some of this structure is less straightforward. It stems from our observation [4] that *in practice*, when mathematicians are *using* theories rather than developing new ones, they tend to work in a rather “flat” name space. An analogy: someone working in Group Theory will unconsciously assume the availability of all concepts from a standard textbook, with their “usual” names and meanings.

As their goal is to get some work done, whatever structure system builders have decided to use to construct their system should not leak into the application domain. They may not be aware of the existence of pointed semigroups, nor should that awareness be forced upon them. Some application domains rely on the “structure of theories”, so we allow those users to see it.

1.2 Contributions

To be explicit, our contributions include:

- A variant of the *category of contexts*, over a dependently-typed type theory as the semantics for theory presentations.
- A simple term language for building theories, using “classical” nomenclature, even though our foundations are unabashedly categorical.
- Using “tiny theories” to allow for maximal reuse and modularity.
- Taking names seriously, since these are meant for human consumption. Moreover, we further emphasize that theory presentations are purely syntactic objects, which are meant to *denote* a semantic object.
- Treating arrows seriously: while this is obvious from a categorical standpoint, it is nevertheless novel in this application.
- Giving multiple (compatible) semantics to our language, which better capture the complete knowledge context of the terms.

1.3 Plan of Paper

We motivate our work with concrete examples in section 2. The theoretical foundations of our work, the fibered category of contexts, is presented in full detail in section 3. This allows us in section 4 to formalize the language of our motivation section, syntactically and semantically. We close with some discussion, related work and conclusions in sections 5–7.

2 Motivation for Theory Presentation Combinators

Let us compare the presentation of two simple theories:

```
Monoid := Theory {
  U: type;  *: (U,U) -> U;  e: U;
  axiom rightIdentity_*_e: forall x:U. x*e = x;
  axiom leftIdentity_*_e: forall x:U. e*x = x;
  axiom associative_*: forall x,y,z:U. (x*y)*z = x*(y*z)}
```

```
CommutativeMonoid := Theory {
  U: type;  *: (U,U) -> U;  e: U;
  axiom rightIdentity_*_e: forall x:U. x*e = x;
  axiom leftIdentity_*_e: forall x:U. e*x = x;
  axiom associative_*: forall x,y,z:U. (x*y)*z = x*(y*z);
  axiom commutative_*: forall x,y:U. x*y = y*x}
```

They are identical, save for the `commutative_*` axiom, as expected. Given `Monoid`, it would be much more economical to define

```
CommutativeMonoid := Monoid extended by {
  axiom commutative_*: forall x,y:U. x*y = y*x}
```

and “expand” this definition, if necessary. Of course, given `Group`, we would similarly find ourselves writing

```
CommutativeGroup := Group extended by {
  axiom commutative_*: forall x,y:U. x*y = y*x}
```

which is also wasteful, as well as dangerous: is this “the same” axiom as before, or a different one? There is no real way to tell. It is natural to further extend our language with a facility that expresses this sharing. Taking a cue from previous work, we might want to say

```
CommutativeGroup := combine CommutativeMonoid, Group over Monoid
```

Informally, this can be read as saying that `Group` and `CommutativeMonoid` are both “extensions” of `Monoid`, and `CommutativeGroup` is formed by the union (amalgamated sum) of those extensions. Another frequent feature is *renaming*: an `AbelianGroup`, while isomorphic to a `CommutativeGroup`, is usually presented additively. We could express this as

```
AbelianGroup := CommutativeGroup [ * |-> +, e |-> 0 ]
```

Unfortunately, while this “works” to build a sizeable library (say of the order of 500 concepts) in a very economical way, it is quite brittle. Let us examine the reasons. It should be clear that by `combine`, we really mean *pushout*¹. But a pushout is a 5-ary operation on 3 objects and 2 arrows; our syntax gives the 3 objects and leaves the arrows implicit. This is a very serious mistake: these arrows are (in general) not easy to infer, especially in the presence of renaming. For example, there are two distinct arrows from `Monoid` to `Ring`, with neither

¹ Following Burstall and Goguen [2] and Smith [18,19] and many others since.

arrow being “better” than the other. Furthermore, we know that pushouts can also be regarded as a 2-ary operation on arrows. In other words, even though our goal is to produce *theory presentations*, our decision to use pushouts² as a fundamental building block gives us no choice but to **take arrows seriously**.

So our task is now to find a category with “theory presentations” as objects, and with arrows which somehow express the notions of extending, combining and renaming as defined above. But before we explore that in depth, let us further examine our operations. First, there is nothing specific to `CommutativeGroup` in the renaming $* \mapsto +, e \mapsto 0$, this can be applied to any theory where the pairs $(*, +)$ and $(e, 0)$ have compatible signatures (including being undefined). Similarly, `extend` really defines a “construction” which can be applied whenever all the symbols used in the extension are defined. In other words, a reasonable semantics should associate a whole class of arrows³ to these operations.

But there is one more aspect to consider: in all our examples above, we have used short, meaningful names. While great for humans, they are in part at fault in the failure of being able to infer arrows. If, like in MMT [16], we used long names, might we be able to build a robust system? Maybe so, but it would immediately fall afoul of our second requirement: irrelevant information such as choices made by developers regarding the order in which to build theories, would leak into the long names, and thus be seen by users. Furthermore, when there is ambiguity, a long name system can indeed resolve that ambiguity, but at too high a cost to humans in absurdly long names for certain concepts.

In other words, to be able to maintain human-readable names for all concepts, we will put the burden on the *library developers* to come up with a reasonable naming scheme, rather than to push that issue onto end users. Another way to see this is that symbol choice carries a lot of intentional, as well as contextual, information which is commonly used in mathematical practice. Thus, to avoid leaking irrelevant information and to maintain intentional/contextual information, we will insist that on **taking names seriously**.

3 Category of Contexts

We observe that theories from the previous section can all be specified as contexts of some dependent type theory. The work in this paper is abstract over the exact details of the dependent type theory⁴ so we simply assume that some dependent type theory is given. Following Cartmell [8], we form the category of contexts \mathbb{C} of the given dependent type theory. The objects of \mathbb{C} are contexts Γ that occur in judgements like $\Gamma \vdash s : \sigma$ of the dependent type theory. A context Γ consists of a sequence of pairs of labels and types (or kinds or propositions),

$$\Gamma := \langle x_0 : \sigma_0; \dots; x_{n-1} : \sigma_{n-1} \rangle,$$

² Which will in fact become pullbacks.

³ We are being deliberately vague here, Section 3 will make this precise.

⁴ In fact, we expect this work to apply not only to dependent type theories, but to any classifying category [14].

such that for each $i < n$ the judgement

$$\langle x_0 : \sigma_0; \dots; x_{i-1} : \sigma_{i-1} \rangle \vdash \sigma_i : \text{Type}$$

holds (resp. $:$ Kind, or $:$ Prop). Contexts of dependent type theory can be used to define the types, operations, relations and axioms of a theory. We will use the abbreviation $\langle x : \sigma \rangle_0^{n-1}$ for a context Γ , and $\text{\textcircled{;}}$ for concatenation of two such sequences.

Example 1. We can define the theory of semigroups via

$$\text{Semigroup} := \left\langle \begin{array}{l} U : \text{Type} \\ (*) : U \times U \rightarrow U \\ \text{associative} : \forall x, y, z : U. (x * y) * z = x * (y * z) \end{array} \right\rangle$$

where we use Haskell-style notation where (\square) indicates (the name of) a binary function used infix in terms.

Normally contexts are considered up to α -equivalence, that is, renaming or permuting the labels of a context makes no difference. But since labels do make a difference, we will not do so. However, α -equivalent *terms* and *types* continue to be considered equivalent.

Example 2. The signature for `AdditiveSemigroup` is given as the context

$$\left\langle \begin{array}{l} U : \text{Type} \\ (+) : U \times U \rightarrow U \\ \text{associative} : \forall x, y, z : U. (x + y) + z = x + (y + z) \end{array} \right\rangle$$

Traditionally `Semigroup` and `AdditiveSemigroup` would be considered the same context because they are α -equivalent.

In the rest of this section, we will use the convention that $\Gamma = \langle x : \sigma \rangle_0^{n-1}$ and $\Delta = \langle y : \tau \rangle_0^{m-1}$. Given two contexts Γ and Δ , a morphism $\Gamma \rightarrow \Delta$ of \mathbb{C} consists of an assignment $[y_0 \mapsto t_0, \dots, y_m \mapsto t_{m-1}]$, abbreviated as $[y \mapsto t]_0^{m-1}$ where the t_0, \dots, t_{m-1} are terms such that

$$\Gamma \vdash t_0 : \tau_0 \quad \dots \quad \Gamma \vdash t_{m-1} : \tau_{m-1} [y \mapsto t]_0^{m-2}$$

all hold, where $\tau [y \mapsto t]_0^i$ denotes the type τ with the labels y_0, \dots, y_i substituted by the corresponding terms of the assignment. We will also use $\text{\textcircled{;}}$ to denote concatenation of assignments, and $[y_{f(j)} \mapsto t_{g(j)}]_{j=a}^b$ for the “obvious” generalized assignment.

Notice that an arrow from Γ to Δ is an assignment from the labels of Δ to terms in Γ . This definition of an arrow may seem backwards at first, but it is defined this way because arrows transform “models” of theories of Γ to “models” of theories of Δ . For example, every Abelian Semigroup is, or rather can be transformed into, an Additive Semigroup by simply forgetting that the

Semigroup is Abelian. A later example [4](#) will give the explicit arrow from Abelian Semigroup to Additive Semigroup that captures this transformation.

Let us fix \mathbb{V} as the (countable) infinite set of labels used in contexts. If $\pi : \mathbb{V} \rightarrow \mathbb{V}$ is a permutation of labels, then we can define an action of this permutation on terms, types and contexts:

$$\pi \cdot \langle x : \sigma \rangle_0^{n-1} := \langle \pi(x_0) : \pi \cdot \sigma_0; \dots; \pi(x_{n-1}) : \pi \cdot \sigma_{n-1} \rangle \equiv \langle \pi x : \pi \cdot \sigma \rangle_0^{n-1}$$

where $\pi \cdot \sigma_i$ is the action induced on the (dependent) type σ_i by renaming labels. The action of π induces an endofunctor $(\pi \cdot -) : \mathbb{C} \rightarrow \mathbb{C}$. Furthermore, each permutation $\pi : \mathbb{V} \rightarrow \mathbb{V}$ induces a natural transformation in $I_\pi : (\pi \cdot -) \Rightarrow \text{id}_{\mathbb{C}}$ where

$$I_\pi(\Gamma) := [x_0 \mapsto \pi(x_0), \dots, x_{n-1} \mapsto \pi(x_{n-1})] : \pi \cdot \Gamma \rightarrow \Gamma.$$

We call an assignment of the form $I_\pi(\Gamma)$ a **renaming**. Because permutations are invertible, each renaming $I_\pi(\Gamma) : \pi \cdot \Gamma \rightarrow \Gamma$ is an isomorphism whose inverse is the renaming $I_{\pi^{-1}}(\pi \cdot \Gamma) : \Gamma \rightarrow \pi \cdot \Gamma$. From this we can see that α -equivalent contexts are isomorphic.

Example 3. Let $\pi : \mathbb{V} \rightarrow \mathbb{V}$ be some permutation such that $\pi(U) = U$, $\pi((*) = (+)$, and $\pi(\text{associative}) = \text{associative}$. By the definition of $I_\pi(\text{Semigroup}) : \text{AdditiveSemigroup} \rightarrow \text{Semigroup}$, we have that

$$I_\pi(\text{Semigroup}) := [U \mapsto U; (*) \mapsto (+); \text{associative} \mapsto \text{associative}]$$

is a renaming isomorphism between the contexts in examples [1](#) and [2](#).

The category of nominal assignments, \mathbb{B} , a sub-category of \mathbb{C} will be quite important for use. For example, theorem [2](#) will show that \mathbb{B} is the base category of a fibration.

Definition 1. *The category of nominal assignments, \mathbb{B} , has the same objects as \mathbb{C} , but only those morphisms whose terms are labels.*

Thus a morphism in \mathbb{B} is an assignment of the form $[y_i \mapsto x_{a(i)}]_{i=0}^{m-1}$ such that the judgements

$$\Gamma \vdash x_{a(0)} : \tau_0 \quad \dots \quad \Gamma \vdash x_{a(m-1)} : \tau_{m-1} \quad [y_i \mapsto x_{a(i)}]_{i=0}^{m-2}$$

all hold.

Definition 2. *We define Γ to be a **sub-context** of Γ^+ if every element $x : \tau$ of Γ occurs in Γ^+ .*

Definition 3. *We call an assignment $A : \Gamma \rightarrow \Delta$ a **diagonal assignment** if A is of the form $[y \mapsto y]_0^{n-1}$ (where $\Delta = \langle y : \tau \rangle_0^{n-1}$), denoted by $\delta_\Delta : \Gamma \rightarrow \Delta$.*

Definition 4. *An assignment $A : \Gamma^+ \rightarrow \Gamma$ is an **extension** when Γ is a sub-context of Γ^+ , and A is the diagonal assignment.*

Notice that an extension points from the extended context to the sub-context. This is the reverse from what Burstall and Goguen [2] use (and most of the algebraic specification community followed their lead). Our direction is inherited from \mathbb{C} , the category of contexts, which is later required by theorem 2 to satisfy the technical definition of a fibration.

Example 4. Consider the theory `AbelianSemigroup` given as

$$\left\langle \begin{array}{l} U : U : \text{Type} \\ (+) : U \times U \rightarrow U \\ \text{associative} : \forall x, y, z : U. (x + y) + z = x + (y + z) \\ \text{commutative} : \forall x, y : U. (x + y) = (y + x) \end{array} \right\rangle$$

Then $\delta_{\text{AdditiveSemigroup}} : \text{AbelianSemigroup} \rightarrow \text{AdditiveSemigroup}$ is an extension.

Example 5. Consider the following two distinct contexts (C_1, C_2) for the theory of left unital Magmas with the order of their operators swapped:

$$\left\langle \begin{array}{l} U : \text{Type} \\ e : U \\ (*) : U \times U \rightarrow U \\ \text{leftIdentity} : \forall x : U. e * x = x \end{array} \right\rangle \quad \left\langle \begin{array}{l} U : \text{Type} \\ (*) : U \times U \rightarrow U \\ e : U \\ \text{leftIdentity} : \forall x : U. e * x = x \end{array} \right\rangle$$

The diagonal assignment $\delta_{C_1} : C_2 \rightarrow C_1$ is an extension (as is $\delta_{C_2} : C_1 \rightarrow C_2$).

Notice that, for any given contexts Γ^+ and Γ , there exists an extension $\Gamma^+ \rightarrow \Gamma$ if and only if Γ is a sub-context of Γ^+ . If Γ is a sub-context of Γ^+ then the diagonal assignment $\delta_\Gamma : \Gamma^+ \rightarrow \Gamma$ is the unique extension.

In general, a renaming $I_\pi : \pi \cdot \Gamma \rightarrow \Gamma$ will not be an extension unless π is the identity on the labels from Γ . In our work, both renaming and extensions are used together, so we want to consider a broader class of nominal assignments that include both extensions and renamings.

Definition 5. *Those nominal assignments where every label occurs at most once will be called general extensions.*

We see that for every permutation of labels $\pi : \mathbb{W} \rightarrow \mathbb{W}$ and every context Γ that $I_\pi(\Gamma) : \pi \cdot \Gamma \rightarrow \Gamma$ is a general extension (and hence also a nominal assignment).

Theorem 1. *Every general extension $A : \Gamma^+ \rightarrow \Delta$ can be turned into an extension by composing it with an appropriate renaming.*

The proof of this theorem, along with all other theorems, lemmas and corollaries in this section can be found in the long version of this paper [7].

Corollary 1. *Every general extension $A : \Gamma^+ \rightarrow \Delta$ can be decomposed into an extension $A_e : \Gamma^+ \rightarrow \Gamma$ followed by a renaming $A_r : \Gamma \rightarrow \Delta$.*

These general extensions form a category which plays an important rôle.

$$\begin{array}{ccc}
 \Gamma^+ & \xrightarrow{f^+} & \Delta^+ \\
 A \downarrow & & \downarrow B \\
 \Gamma & \xrightarrow{f^-} & \Delta
 \end{array}$$

Definition 6. *The category of general extensions \mathbb{E} has all general extensions from \mathbb{B} as objects, and given two general extensions $A : \Gamma^+ \rightarrow \Gamma$ and $B : \Delta^+ \rightarrow \Delta$, an arrow $f : A \rightarrow B$ is a commutative square from \mathbb{B} . We will denote this commutative square by $\langle f^+, f^- \rangle : A \rightarrow B$.*

We remind the reader of the usual convention in category theory where arrows include their domain and codomain as part of their structure (which we implicitly use in the definition above).

Lemma 1. *Every general extension is isomorphic in \mathbb{E} to an extension $B : \Gamma^\circ \rightarrow \Gamma$ where Γ is an initial segment of Γ° .*

This category of general extensions \mathbb{E} is fibered over the category \mathbb{B} by the codomain functor $\text{cod} : \mathbb{E} \rightarrow \mathbb{B}$. Given general extensions $A : \Gamma^+ \rightarrow \Gamma$ and $B : \Delta^+ \rightarrow \Delta$ and a morphism $\langle f^+, f^- \rangle : A \rightarrow B$ in \mathbb{E} we have

$$\text{cod}(A) := \Gamma \qquad \text{cod}(f) := f^-$$

Theorem 2. *The functor $\text{cod} : \mathbb{E} \rightarrow \mathbb{B}$ is a fibration.*

Corollary 2. *Given $u : \Gamma \rightarrow \Delta$, a general extension $A : \Delta^+ \rightarrow \Delta$, and a cartesian lifting $\bar{u}(A) : u^*(A) \rightarrow A$, if u is a general extension, then $\bar{u}(A)^+$ is also a general extension.*

Example 6. The nominal assignment (and general extension)

$$u := \left[\begin{array}{l} U \mapsto U \\ (*) \mapsto (+) \\ \text{associative} \mapsto \text{associative} \end{array} \right] : \text{AbelianSemigroup} \rightarrow \text{Semigroup}$$

and the extension $A := \delta_{\text{Semigroup}} : \text{Monoid} \rightarrow \text{Semigroup}$ induce the existence (via theorem 2) of some Cartesian lifting $\bar{u}(A) : u^*(A) \rightarrow A$ in \mathbb{E} . One example of such a Cartesian lifting for \bar{u} is

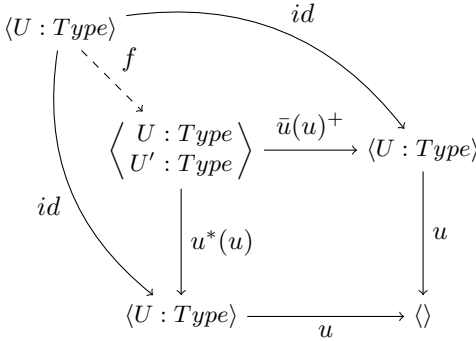
$$\begin{array}{ccc}
 \text{AbelianMonoid} & \xrightarrow{\quad} & \text{Monoid} \\
 \downarrow u^*(A) & \bar{u}(A)^+ & \downarrow A \\
 \text{AbelianSemigroup} & \xrightarrow{u} & \text{Semigroup}
 \end{array}$$

where `AbelianMonoid` is

$$\left\langle \begin{array}{l} U : \text{Type} \\ 0 : U \\ (+) : U \times U \rightarrow U \\ \text{rightIdentity} : \forall x : U. x + 0 = x \\ \text{leftIdentity} : \forall x : U. 0 + x = x \\ \text{associative} : \forall x, y, z : U. (x + y) + z = x + (y + z) \\ \text{commutative} : \forall x, y : U. (x + y) = (y + x) \end{array} \right\rangle$$

and $u^*(A) : \text{AbelianMonoid} \rightarrow \text{AbelianSemigroup}$ is the diagonal assignment, where $\bar{u}(A)^+ : \text{AbelianMonoid} \rightarrow \text{Monoid}$ is

$$\bar{u}(A)^+ := \left[\begin{array}{l} U \mapsto U \\ e \mapsto 0 \\ (*) \mapsto (+) \\ \text{rightIdentity} \mapsto \text{rightIdentity} \\ \text{leftIdentity} \mapsto \text{leftIdentity} \\ \text{associative} \mapsto \text{associative} \end{array} \right]$$



a Cartesian lifting of u over itself. The mediating arrow for $\text{id} : \langle U : \text{Type} \rangle \rightarrow \langle U : \text{Type} \rangle$ and itself must be

$$\begin{aligned} f &: \langle U : \text{Type} \rangle \rightarrow \langle U : \text{Type}; U' : \text{Type} \rangle \\ f &:= [U \mapsto U, U' \mapsto U] \end{aligned}$$

which is not a general extension.

4 Semantics of Theory Presentation Combinators

Like in the previous section, we will assume that we have a background type theory with well-formedness *judgments*, which defines four different sorts, namely (`Type`, `Term`, `Kind`, `Prop`). The symbols used in the type theory itself will be called

In almost all of the development of the algebraic hierarchy, the nominal assignments that we use are all general extensions. However, it is important to note that the definition of a Cartesian lifting requires nominal assignments that are not necessarily general extensions, even if all the inputs are general extensions.

Consider the simple case (pictured above) where $u : \langle U : \text{Type} \rangle \rightarrow \langle \rangle$ is the unique extension, and

labels, whereas the symbols used for theory presentations will be called *names*. As above, $a \mapsto b$ denotes a *substitution*. Using this, we can define the formal syntax for our combinators as follows.

$a, b, c \in \text{labels}$	$\tau \in \text{Type}$	$\text{tpc} ::= \text{extend } A \text{ by } \{l\}$
$A, B, C \in \text{names}$	$k \in \text{Kind}$	$\text{combine } A \ r_1, \ B \ r_2$
$l \in \text{judgments}^*$	$t \in \text{Term}$	$A ; B$
$r \in (a_i \mapsto b_i)^*$	$\theta \in \text{Prop}$	$A \ r$
		Empty
		$\text{Theory } \{l\}$

Intuitively, the six forms correspond to: extending a theory with new knowledge, combining two theories into a larger one, sequential composition of theories, renaming, a constant for the Empty theory, and an explicit theory.

What we do next is slightly unusual: rather than give a single denotational semantics, we will give *two*, one in terms of objects of \mathbb{B} , and one in terms of objects of \mathbb{E} (which are special arrows in \mathbb{B}). In fact, we have a third semantics, in terms of (partial) Functors over the contextual category, but we will omit it for lack of space. First, we give the semantics in terms of objects of \mathbb{B} , where $\llbracket - \rrbracket_\pi$ is the (obvious) semantics in $\mathbb{V} \rightarrow \mathbb{V}$ of a renaming.

$$\begin{array}{l}
 \llbracket - \rrbracket_{\mathbb{B}} : \text{tpc} \rightarrow |\mathbb{B}| \\
 \llbracket \text{Empty} \rrbracket_{\mathbb{B}} = \langle \rangle \\
 \llbracket \text{Theory } \{l\} \rrbracket_{\mathbb{B}} \cong \langle l \rangle \\
 \llbracket A \ r \rrbracket_{\mathbb{B}} = \llbracket r \rrbracket_\pi \cdot \llbracket A \rrbracket_{\mathbb{B}} \\
 \llbracket A ; B \rrbracket_{\mathbb{B}} = \llbracket B \rrbracket_{\mathbb{B}} \\
 \llbracket \text{extend } A \text{ by } \{l\} \rrbracket_{\mathbb{B}} \cong \llbracket A \rrbracket_{\mathbb{B}} \ ; \ \langle l \rangle \\
 \llbracket \text{combine } A_1 r_1, A_2 r_2 \rrbracket_{\mathbb{B}} \cong D
 \end{array}$$

$$\begin{array}{ccc}
 D & \xrightarrow{\llbracket r_1 \rrbracket_\pi \circ \delta_{A_1}} & A_1 \\
 \downarrow \llbracket r_2 \rrbracket_\pi \circ \delta_{A_2} & & \downarrow \delta_A \\
 A_2 & \xrightarrow{\delta_A} & A
 \end{array}$$

where D comes from the (potential) pullback diagram on the right, in which we omit $\llbracket - \rrbracket_{\mathbb{B}}$ around the A s for clarity. We use \cong to abbreviate “when the rhs is a well-formed context”. For the semantics of **combine**, it must be the case where the diagram at right is a *pullback* (in \mathbb{B}), where A is the greatest lower bound context $\llbracket A_1 \rrbracket_{\mathbb{B}} \sqcap \llbracket A_2 \rrbracket_{\mathbb{B}}$. Furthermore $\llbracket r_1 \rrbracket_\pi$ and $\llbracket r_2 \rrbracket_\pi$ must leave A invariant. We remind the reader of the requirement for these renamings: the users must pick which cartesian lifting they want, and this cannot be done automatically (as demonstrated at the end of last section).

The second semantics, is in terms of the objects of \mathbb{E} , in other words, the *special* arrows of \mathbb{B} , as defined in Section 3.

$$\begin{aligned}
 & \llbracket - \rrbracket_{\mathbb{E}} : \text{tpc} \rightarrow |\mathbb{E}| \\
 & \llbracket \text{Empty} \rrbracket_{\mathbb{E}} = \text{id}_{\langle \rangle} \\
 & \llbracket \text{Theory } \{l\} \rrbracket_{\mathbb{E}} \cong !_{\langle l \rangle} \\
 & \llbracket A \ r \rrbracket_{\mathbb{E}} = \llbracket r \rrbracket_{\pi} \cdot \llbracket A \rrbracket_{\mathbb{E}} \\
 & \llbracket A; B \rrbracket_{\mathbb{E}} = \llbracket A \rrbracket_{\mathbb{E}} \circ \llbracket B \rrbracket_{\mathbb{E}} \\
 & \llbracket \text{extend } A \text{ by } \{l\} \rrbracket_{\mathbb{E}} \cong \delta_A \\
 & \llbracket \text{combine } A_1 r_1, A_2 r_2 \rrbracket_{\mathbb{E}} \cong \llbracket r_1 \rrbracket_{\pi} \circ \delta_{T_1} \circ \llbracket A_1 \rrbracket_{\mathbb{E}} \\
 & \qquad \qquad \qquad \cong \llbracket r_2 \rrbracket_{\pi} \circ \delta_{T_2} \circ \llbracket A_2 \rrbracket_{\mathbb{E}}
 \end{aligned}$$

$$\begin{array}{ccc}
 D & \xrightarrow{\llbracket r_1 \rrbracket_{\pi} \circ \delta_{T_1}} & T_1 \\
 \downarrow \llbracket r_2 \rrbracket_{\pi} \circ \delta_{T_2} & & \downarrow A_1 \\
 T_2 & \xrightarrow{A_2} & T
 \end{array}$$

The diagram on the right has to be verified to be a pullback diagram (this is why the semantics is partial here too). Here we assume $\llbracket A_1 \rrbracket_{\mathbb{E}} \in \text{Hom}(T_1, T)$ and $\llbracket A_2 \rrbracket_{\mathbb{E}} \in \text{Hom}(T_2, T)$, and that both $\llbracket r_1 \rrbracket_{\pi}$ and $\llbracket r_2 \rrbracket_{\pi}$ leave T invariant.

Theorem 3. *For all tpc terms except combine, $\llbracket s \rrbracket_{\mathbb{B}} = \text{dom} \llbracket s \rrbracket_{\mathbb{E}}$. When $s \equiv \text{combine } A_1 r_1, A_2 r_2$, if $\text{cod}(\llbracket A_1 \rrbracket_{\mathbb{E}}) = \text{cod}(\llbracket A_2 \rrbracket_{\mathbb{E}}) = \llbracket A_1 \rrbracket_{\mathbb{B}} \sqcap \llbracket A_2 \rrbracket_{\mathbb{B}}$, and neither arrows $\llbracket A_1 \rrbracket_{\mathbb{E}}$ nor $\llbracket A_2 \rrbracket_{\mathbb{E}}$ involve renamings, then $\llbracket s \rrbracket_{\mathbb{B}} = \text{dom} \llbracket s \rrbracket_{\mathbb{E}}$ in that case as well.*

The proof is a straightforward comparison of the semantic equations. This theorem basically says that, as long as we only use `combine` on the “natural” base of two arrows which are pure extensions, our semantics are compatible. In a *tiny theories* setting, this can be arranged.

5 Discussion

It is important to note that we are essentially parametric in the underlying type theory. This should allow us to be able to generalize our work in ways similar to Kohlhase and Rabe’s MMT [16].

The careful reader might have notice that in the syntax of section 2, our `combine` had an `over` keyword. This allowed our previous implementation [5] to come partway to the \mathbb{E} semantics above. This is a straightforward extension to the semantics: $\llbracket \text{combine } A_1 r_1, A_2 r_2 \text{ over } C \rrbracket_{\mathbb{B}}$ would replace $A = \llbracket A_1 \rrbracket_{\mathbb{B}} \sqcap \llbracket A_2 \rrbracket_{\mathbb{B}}$ with $\llbracket C \rrbracket_{\mathbb{B}}$, with corresponding adjustments to the rest of the pushout diagram. For $\llbracket - \rrbracket_{\mathbb{E}}$, one would insist that $\text{cod}(\llbracket A_1 \rrbracket_{\mathbb{E}}) = \text{cod}(\llbracket A_2 \rrbracket_{\mathbb{E}}) = \llbracket C \rrbracket_{\mathbb{B}}$.

What is more promising [5] still is that most of our terms can also be interpreted as *Functors* between fibered categories. This gives us a semantic for each term as a “construction”, which can be reused (as in our example with commutativity in section 2). Furthermore, since fibered categories interact well with limits and colimits, we should also be able to combine constructions and diagrams so as to fruitfully capture further structure in theory hierarchies.

It should also be noted that our work also extends without difficulty to having *definitions* (and other such conservative extensions) in our contexts. This is

⁵ Work in progress.

especially useful when transporting theorems from one setting to another, as is done when using the “Little Theories” method [10]. We also expect our work to extend to allow Cartesian liftings of extensions over arbitrary assignments (aka views) from the full category of contexts.

Lastly, we have implemented a “flattener” for our semantics, which basically turns a presentation A into a flat presentation $\text{Theory}\{l\}$ by computing $\text{cod}(\llbracket A \rrbracket_{\mathbb{E}})$. This fulfils our second requirement, where the method of construction of a theory is invisible to users of flat theories.

6 Related Work

We will not consider work in universal algebra, institutions or categorical logic as “related”, since they employ α -equivalence on labels (as well as on bound variables), which we consider un-helpful for theory presentations meant for human consumption. We also leave aside much interesting work on dependent record types (which we use), as these are but one implementation method for theories, and we consider *contexts* as a much more fundamental object.

We have been highly influenced by the early work of Burstall and Goguen [23], and Doug Smith’s Specware [18,19]. They gave us the basic semantic tools we needed. But we quickly found out, much to our dismay, that neither of these approaches seemed to scale very well. Later, we were hopeful that CASL [9] might work for us, but then found that their own base library was improperly factored and full of redundancies. Of the vast algebraic specification literature around this topic, we want to single out the work of Oriat [15] on isomorphism of specification graphs as capturing similar ideas to ours on extreme modularity. And it cannot be emphasized enough how crucial Bart Jacob’s book [14] has been to our work.

From the mathematical knowledge management side, it should be clear that MMT [16] is closely related. The main differences are that they are quite explicit about being foundations-independent (it is implicit in our work), they use long names, and their theory operations are mostly theory-internal, while ours are external. This makes a big difference, as it allows us to have multiple semantics, while theirs has to be fixed. And, of course, the work presented in the current paper covers just a small part of the vast scope of MMT.

There are many published techniques and implementations of algebraic hierarchies in dependently typed proof assistants including [12,20,11,17]. Our work does not compete with these implementations, but rather complements them. More specifically, we envision our work as a meta-language which can be used to specify algebraic hierarchies, which can subsequently be implemented by using any of the aforementioned techniques. In particular we note that maintaining the correct structures for packed-classes of [11] is particularly difficult, and deriving the required structures from a hierarchy specification would alleviate much of this burden. Other cited work, (for example [17]) focus on other difficult problems such as *usability*, via providing coercions and unification hints to match particular terms to theories. Even though some similar techniques (categorical pullbacks) are used in a similar context, the details are very different.

7 Conclusion

There has been a lot of work done in mathematics to give structure to mathematical theories, first via universal algebra, then via category theory (e.g. Lawvere theories). But even though a lot of this work started out being somewhat syntactic, very quickly it became mostly semantic, and thus largely useless for the purposes of concrete implementations.

We make the observation that, with a rich enough type theory, we can identify the category of theory presentations with the opposite of the category of contexts. This allows us to draw freely from developments in categorical logic, as well as to continue to be inspired by algebraic specifications. Interestingly, key here is to make the opposite choice as Goguen's in two ways: our base language is firmly higher-order, while our "module" language is first-order, and we work in the opposite category.

We provide a simple-to-understand term language of "theory expression combinators", along with multiple (categorical) semantics. We have shown that these fit our requirements of allowing to capture mathematical structure, while also allowing this structure to be hidden from users.

Even more promising, our use of very standard categorical constructions points the way to simple generalizations which should allow us to capture even more structure, without having to rewrite our library. Furthermore, as we are independent of the details of the type theory, this structure seems very robust, and our combinators should thus port easily to other systems.

References

1. Asperti, A., Sacerdoti Coen, C.: Some Considerations on the Usability of Interactive Provers. In: Autexier, S., Calmet, J., Delahaye, D., Ion, P.D.F., Rideau, L., Rioboo, R., Sexton, A.P. (eds.) AISC 2010. LNCS, vol. 6167, pp. 147–156. Springer, Heidelberg (2010), <http://dl.acm.org/citation.cfm?id=1894483.1894498>
2. Burstall, R.M., Goguen, J.A.: Putting theories together to make specifications. In: IJCAI, pp. 1045–1058 (1977)
3. Burstall, R.M., Goguen, J.A.: The Semantics of Clear, a Specification Language. In: Bjorner, D. (ed.) Abstract Software Specifications. LNCS, vol. 86, pp. 292–332. Springer, Heidelberg (1980)
4. Carette, J., Farmer, W.M.: High-Level Theories. In: Autexier, S., Campbell, J., Rubio, J., Sorge, V., Suzuki, M., Wiedijk, F. (eds.) AISC 2008, Calculemus 2008, and MKM 2008. LNCS (LNAI), vol. 5144, pp. 232–245. Springer, Heidelberg (2008)
5. Carette, J., Farmer, W.M., Jeremic, F., Maccio, V., O'Connor, R., Tran, Q.: The mathscheme library: Some preliminary experiments. Tech. rep., University of Bologna, Italy (2011), uBLCS-2011-04
6. Carette, J., Kiselyov, O.: Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. *Sci. Comput. Program.* 76(5), 349–375 (2011)
7. Carette, J., O'Connor, R.: Theory Presentation Combinators (2012), <http://arxiv.org/abs/1204.0053v2>

8. Cartmell, J.: Generalised algebraic theories and contextual categories. *Annals of Pure and Applied Logic* 32, 209–243 (1986), <http://www.sciencedirect.com/science/article/pii/0168007286900539>
9. CoFI (The Common Framework Initiative): *CasI Reference Manual*. LNCS, IFIP Series, vol. 2960. Springer (2004)
10. Farmer, W.M., Guttman, J.D., Thayer, F.J.: *Little Theories*. In: Kapur, D. (ed.) *CADE 1992*. LNCS, vol. 607, pp. 567–581. Springer, Heidelberg (1992)
11. Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: *Packaging Mathematical Structures*. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) *TPHOLs 2009*. LNCS, vol. 5674, pp. 327–342. Springer, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-03359-9_23
12. Geuvers, H., Wiedijk, F., Zwanenburg, J.: *A Constructive Proof of the Fundamental Theorem of Algebra without Using the Rationals*. In: Callaghan, P., Luo, Z., McKinna, J., Pollack, R. (eds.) *TYPES 2000*. LNCS, vol. 2277, pp. 96–111. Springer, Heidelberg (2002)
13. Grabowski, A., Schwarzweiler, C.: *On Duplication in Mathematical Repositories*. In: Autexier, S., Calmet, J., Delahaye, D., Ion, P., Rideau, L., Rioboo, R., Sexton, A. (eds.) *AISC 2010*. LNCS, vol. 6167, pp. 300–314. Springer, Heidelberg (2010)
14. Jacobs, B.: *Categorical Logic and Type Theory*. *Studies in Logic and the Foundations of Mathematics*, vol. 141. North Holland, Amsterdam (1999)
15. Oriat, C.: *Detecting equivalence of modular specifications with categorical diagrams*. *Theor. Comput. Sci.* 247(1-2), 141–190 (2000)
16. Rabe, F., Kohlhase, M.: *A Scalable Module System*, <http://kwarc.info/frabe/Research/mmt.pdf>
17. Sacerdoti Coen, C., Tassi, E.: *Nonuniform coercions via unification hints*. In: Hirschowitz, T. (ed.) *TYPES*. *EPTCS*, vol. 53, pp. 16–29 (2009)
18. Smith, D.R.: *Constructing specification morphisms*. *Journal of Symbolic Computation* 15, 5–6 (1993)
19. Smith, D.R.: *Mechanizing the development of software*. In: Broy, M., Steinbrueggen, R. (eds.) *Calculational System Design, Proceedings of the NATO Advanced Study Institute*, pp. 251–292. IOS Press, Amsterdam (1999)
20. Spitters, B., van der Weegen, E.: *Type classes for mathematics in type theory*. *Mathematical Structures in Computer Science* 21(4), 795–825 (2011)
21. Wiedijk, F.: *Estimating the cost of a standard library for a mathematical proof checker* (2001), <http://www.cs.ru.nl/~freek/notes/mathstdlib2.pdf>

Verifying an Algorithm Computing Discrete Vector Fields for Digital Imaging*

Jónathan Heras, María Poza, and Julio Rubio

Department of Mathematics and Computer Science of University of La Rioja
{jonathan.heras,maria.poza,julio.rubio}@unirioja.es

Abstract. In this paper, we present a formalization of an algorithm to construct *admissible discrete vector fields* in the COQ theorem prover taking advantage of the SSREFLECT library. Discrete vector fields are a tool which has been welcomed in the *homological analysis* of digital images since it provides a procedure to reduce the amount of information but preserving the homological properties. In particular, thanks to discrete vector fields, we are able to compute, inside COQ, homological properties of *biomedical images* which otherwise are out of the reach of this system.

Keywords: Discrete Vector Fields, Haskell, COQ, SSREFLECT, Integration.

1 Introduction

Kenzo [10] is a Computer Algebra System devoted to Algebraic Topology which was developed by F. Sergeraert. This system has computed some homology and homotopy groups which cannot be easily obtained by theoretical or computational means; some examples can be seen in [23]. Therefore, in this situation, it makes sense to analyze the Kenzo programs in order to ensure the correctness of the mathematical results which are obtained thanks to it. To this aim, two different research lines were launched some years ago to apply formal methods in the study of Kenzo.

On the one hand, the ACL2 theorem prover has been used to verify the correctness of actual Kenzo *programs*, see [17,21]. ACL2 fits perfectly to this task since Kenzo is implemented in Common Lisp [14], the same language in which ACL2 is built on. Nevertheless, since the ACL2 logic is first-order, the full verification of Kenzo is not possible, because it uses intensively higher order functional programming. On the other hand, some instrumental Kenzo *algorithms*, involving higher-order logic, have been formalized in the proof assistants Isabelle/HOL and Coq. Namely, we can highlight the formalizations of the Basic Perturbation Lemma in Isabelle/HOL, see [2], and the Effective Homology of Bicomplexes in Coq, published in [9].

* Partially supported by Ministerio de Educación y Ciencia, project MTM2009-13842-C02-01, and by the European Union's 7th Framework Programme under grant agreement nr. 243847 (ForMath).

The work presented in this paper goes in the same direction that the latter approach, formalizing Kenzo *algorithms*. In particular, we have focused on the formalization of *Discrete Vector Fields*, a powerful notion which will play a key role in the new version of Kenzo; see the Kenzo web page [10]. To carry out this task, we will use the COQ proof assistant [7] and its SSREFLECT library [13].

The importance of Discrete Vector Fields, which were first introduced in [11], stems from the fact that they can be used to considerably reduce the amount of information of a discrete object but preserving homological properties. In particular, we can use discrete vector fields to deal with biomedical images inside COQ in a reasonable amount of time.

The rest of this paper is organized as follows. In the next section, we introduce some mathematical preliminaries, which are encoded abstractly in COQ in Section 3. Such an abstract version is *refined* to an effective one in Section 4; namely, the implementation and formal verification of the main algorithm involved in our developments are presented there. In order to ensure the feasibility of our programs, a major issue when applying formal methods, we use them to study a biomedical problem in Section 5. The paper ends with a section of Conclusions and Further work, and the Bibliography.

The interested reader can consult the complete development in <http://wiki.portal.chalmers.se/cse/pmwiki.php/ForMath/ProofExamples#wp3ex5>.

2 Mathematics to Formalize

In this section, we briefly provide the minimal mathematical background needed to understand the rest of the paper. We mainly focus on definitions which, mainly, come from the algebraic setting of discrete Morse theory presented in [24] and the Effective Homology theory [25]. We assume as known the notions of *ring*, *module* over a ring and *module morphism* (see, for instance, [18]).

First of all, let us introduce one of the main notions in the context of Algebraic Topology: *chain complexes*.

Definition 1. A *chain complex* C_* is a pair of sequences $(C_n, d_n)_{n \in \mathbb{Z}}$ where for every $n \in \mathbb{Z}$, C_n is an \mathcal{R} -module and $d_n : C_n \rightarrow C_{n-1}$ is a module morphism, called the *differential map*, such that the composition $d_n d_{n+1}$ is null. In many situations the ring \mathcal{R} is either the integer ring, $\mathcal{R} = \mathbb{Z}$, or the field \mathbb{Z}_2 . In the rest of this section, we will work with \mathbb{Z} as ground ring; later on, we will change to \mathbb{Z}_2 .

The module C_n is called the module of *n-chains*. The image $B_n = \text{im } d_{n+1} \subseteq C_n$ is the (sub)module of *n-boundaries*. The kernel $Z_n = \text{ker } d_n \subseteq C_n$ is the (sub)module of *n-cycles*.

Given a chain complex $C_* = (C_n, d_n)_{n \in \mathbb{Z}}$, the identities $d_{n-1} \circ d_n = 0$ mean the inclusion relations $B_n \subseteq Z_n$: every boundary is a cycle (the converse in general is not true). Thus the next definition makes sense.

Definition 2. The *n-homology group* of C_* , denoted by $H_n(C_*)$, is defined as the quotient $H_n(C_*) = Z_n/B_n$

Chain complexes have a corresponding notion of morphism.

Definition 3. Let $C_* = (C_n, d_n)_{n \in \mathbb{Z}}$ and $D_* = (D_n, \widehat{d}_n)_{n \in \mathbb{Z}}$ be two chain complexes. A *chain complex morphism* $f : C_* \rightarrow D_*$ is a family of module morphisms, $f = \{f_n : C_n \rightarrow D_n\}_{n \in \mathbb{Z}}$, satisfying for every $n \in \mathbb{Z}$ the relation $f_{n-1}d_n = \widehat{d}_n f_n$. Usually, the sub-indexes are skipped, and we just write $fd_C = d_D f$.

Now, we can introduce one of the fundamental notions in the effective homology theory.

Definition 4. A *reduction* ρ between two chain complexes C_* and D_* , denoted in this paper by $\rho : C_* \rightrightarrows D_*$, is a triple $\rho = (f, g, h)$ where $f : C_* \rightarrow D_*$ and $g : D_* \rightarrow C_*$ are chain complex morphisms, $h = \{h_n : C_n \rightarrow C_{n+1}\}_{n \in \mathbb{Z}}$ is a family of module morphism, and the following relations are satisfied:

- 1) $f \circ g = Id_{D_*}$;
- 2) $d_C \circ h + h \circ d_C = Id_{C_*} - g \circ f$;
- 3) $f \circ h = 0$; $h \circ g = 0$; $h \circ h = 0$.

The importance of reductions lies in the fact that given a reduction $\rho : C_* \rightrightarrows D_*$, then $H_n(C_*)$ is isomorphic to $H_n(D_*)$ for every $n \in \mathbb{Z}$. Very frequently, D_* is a much smaller chain complex than C_* , so we can compute the homology groups of C_* much faster by means of those of D_* .

Let us state now the main notions coming from the algebraic setting of Discrete Morse Theory [24].

Definition 5. Let $C_* = (C_n, d_n)_{n \in \mathbb{Z}}$ be a free chain complex with distinguished \mathbb{Z} -basis $\beta_n \subset C_n$. A *discrete vector field* V on C_* is a collection of pairs $V = \{(\sigma_i; \tau_i)\}_{i \in I}$ satisfying the conditions:

- Every σ_i is some element of β_n , in which case $\tau_i \in \beta_{n+1}$. The degree n depends on i and in general is not constant.
- Every component σ_i is a *regular face* of the corresponding τ_i (regular face means that the coefficient of σ_i in $d_{n+1}\tau_i$ is 1 or -1).
- Each generator (*cell*) of C_* appears at most one time in V .

It is not compulsory all the cells of C_* appear in the vector field V .

Definition 6. A cell χ which does not appear in a discrete vector field $V = \{(\sigma_i; \tau_i)\}_{i \in I}$ is called a *critical cell*.

From a discrete vector field on a chain complex, we can introduce V -paths.

Definition 7. A V -path of degree n and length m is a sequence $((\sigma_{i_k}, \tau_{i_k}))_{0 \leq k < m}$ satisfying:

- Every pair $((\sigma_{i_k}, \tau_{i_k}))$ is a component of V and τ_{i_k} is a n -cell.
- For every $0 < k < m$, the component σ_{i_k} is a face of $\tau_{i_{k-1}}$ (the coefficient of σ_{i_k} in $d_n \tau_{i_{k-1}}$ is non-null) different from $\sigma_{i_{k-1}}$.

Now we can present the notion of *admissible* discrete vector field on a chain complex, a concept which can be understood as a *recipe* indicating both the “useless” elements of the chain complex (in the sense, that they can be removed without changing its homology) and the *critical* ones (those whose removal modifies the homology).

Definition 8. A discrete vector field V is *admissible* if for every $n \in \mathbb{Z}$, a function $\lambda_n : \beta_n \rightarrow \mathbb{N}$ is provided satisfying the following property: every V -path starting from $\sigma \in \beta_n$ has a length bounded by $\lambda_n(\sigma)$.

Finally, we can state the theorem where Discrete Morse Theory and Effective Homology converge.

Theorem 9. [24, Theorem 19] Let $C_* = (C_n, d_n)_{n \in \mathbb{Z}}$ be a free chain complex and V be an admissible discrete vector field on C_* . Then the vector field V defines a canonical reduction $\rho : (C_n, d_n) \Rightarrow (C_n^c, d_n^c)$ where $C_n^c = \mathbb{Z}[\beta_n^c]$ is the free \mathbb{Z} -module generated by β_n^c , the critical n -cells.

Therefore, as the bigger the admissible discrete vector field V the smaller the chain complex C_*^c , we need algorithms which produce admissible discrete vector fields as large as possible.

If we consider the case of *finite type* chain complexes, where there is a finite number of generators in each dimension of the chain complex, the differential maps can be represented as matrices. In that case, the problem of finding an admissible discrete vector field on the chain complex can be solved through the computation of an admissible vector field for those matrices.

Definition 10. Let M be a matrix with coefficients in \mathbb{Z} , and with m rows and n columns. A discrete vector field V for this matrix is a set of natural pairs $\{(a_i, b_i)\}$ satisfying these conditions:

1. $1 \leq a_i \leq m$ and $1 \leq b_i \leq n$.
2. The entry $M[a_i, b_i]$ of the matrix is ± 1 .
3. The indexes a_i (resp. b_i) are pairwise different.

Given V be a vector field for our matrix M , we need to know if V is admissible. If $1 \leq a, a' \leq m$, with $a \neq a'$, we can decide $a > a'$ if there is an elementary V -path from a to a' , that is, if a vector (a, b) is present in V and the entry $M[a', b]$ is non-null. In this way, a binary relation is obtained. Then the vector field V is admissible if and only if this binary relation generates a partial order, that is, if there is no loop $a_1 > a_2 > \dots > a_k = a_1$.

Eventually, given a matrix and an admissible discrete vector field on it, we can construct a new matrix, smaller than the original one, preserving the homological properties. This is the equivalent version of Theorem 9 for matrices, a detailed description of the process can be seen in [24, Proposition 14].

In the rest of the paper, we will focus on the formally certified construction of an admissible discrete vector field from a matrix. The task of verifying the reduction process remains as further work.

3 A Non Deterministic Algorithm in SSREFLECT

First of all, we have provided in COQ/SSREFLECT an abstract formalization of admissible discrete vector fields on matrices and a non deterministic algorithm to construct an admissible discrete vector field from a matrix¹. SSREFLECT is an extension for the COQ proof assistant, which was developed by G. Gonthier while formalizing the Four Color Theorem [12]. Nowadays, it is used in the formal proof of the Feit-Thompson theorem [1].

SSREFLECT provides all the necessary tools to achieve our goal. In particular, we take advantage of the `matrix`, `ssralg` and `fingraph` libraries, which formalize, respectively, matrix theory, the main algebraic structures and the theory of finite graphs.

First of all, we are going to define admissible discrete vector field on a matrix M with coefficients in a ring \mathcal{R} , and with m rows and n columns. It is worth noting that our matrices are defined over a generic ring instead of working with coefficients in \mathbb{Z} since the SSREFLECT implementation of \mathbb{Z} , see [6], is not yet included in the SSREFLECT distributed version. The vector fields are represented by a sequence of pairs where the first component is an ordinal m and the second one an ordinal n .

Variable `R` : ringType.

Variables `m n` : nat.

Definition `vectorfield` := seq ('I_m * 'I_n).

Now, we can define in a straightforward manner a function, called `dvf`, which given a matrix M (with coefficients in a ring \mathcal{R} , and with m rows and n columns, `'M[R]_(m,n)`) and an object V of type `vectorfield` checks whether V satisfies the properties of a discrete vector field on M (Definition 10).

Definition `dvf` (`M` : `'M[R]_(m,n)`) (`V` : `vectorfield`) :=
 all [pred p | (M p.1 p.2 == 1) || (M p.1 p.2 == -1)] V &&
 (uniq (map (@fst _ _) V) && uniq (map (@snd _ _) V)).

It is worth noting that the first condition of Definition 10 is implicit in the `vectorfield` type. Now, as we have explained at the end of the previous section, from a discrete vector field V a binary relation is obtained between the first elements of each pair of V . Such a binary relation will be encoded by means of an object of the following type.

Definition `orders` := (simpl_rel 'I_m).

Finally, we can define a function, which is called `advf`, that given a matrix `'M[R]_(m,n)`, M , a `vectorfield`, V and an `orders`, `ords`, as input, tests whether both V satisfies the properties of a discrete vector field on M and the admissibility property for the relations, `ords`, associated with the vector field, V . In order to test the admissibility property we generate the transitive closure of `ords`, using

¹ Thanks are due to Maxime Dénès and Anders Mörtberg which guided us in this development.

the `connect` operator of the `fingraph` library, and subsequently check that there is not any path between the first element of a pair of V and itself.

```
Definition advf (M: 'M[R]_(m,n)) (V:vectorfield) (ords:orders) :=
  dvf M V && all [pred i | ~ (connect ords i i)] (map (@fst _ _) V).
```

Now, let us define a non deterministic algorithm which construct an admissible discrete vector field from a matrix. Firstly, we define a function, `gen_orders`, which generates the relations between the elements of the discrete vector field as we have explained at the end of the previous section.

```
Definition gen_orders (MO : 'M[R]_(m,n)) (i: 'I_m) j :=
  [rel i x | (x != i) && (MO x j != 0)].
```

Subsequently, the function, `gen_adm_dvf`, which generates an admissible discrete vector field from a matrix is introduced. This function invokes a recursive function, `genDvfOrders`, which in each step adds a new component to the vector field in such a way that the admissibility property is fulfilled. The recursive algorithm stops when either there is not any new element whose inclusion in the vector field preserves the admissibility property or the maximum number of elements of the discrete vector field (which is the minimum between the number of columns and the number of rows of the matrix) is reached.

```
Fixpoint genDvfOrders M V (ords : simpl_rel _) k :=
  if k is l.+1 then
    let P := [pred ij | admissible (ij::V) M
                    (relU ords (gen_orders M ij.1 ij.2))] in
    if pick P is Some (i,j)
      then genDvfOrders M ((i,j)::V)
                    (relU ords (gen_orders M i j)) l
    else (V, ords)
  else (V, ords).
```

```
Definition gen_adm_dvf M :=
  genDvfOrders M [::] [rel x y | false] (minn m n).
```

Eventually, we can certify in a straightforward manner (just 4 lines) the correctness of the function `gen_adm_dvf`.

```
Lemma admissible_gen_adm_dvf m n (M : 'M[R]_(m,n)) :
  let (vf,ords) := gen_adm_dvf M in admissible vf M ords.
```

As a final remark, it is worth noting that the function `gen_adm_dvf` is not executable. On the one hand, `SSREFLECT` matrices are locked in a way that do not allow direct computations since they may trigger heavy computations during deduction steps. On the other hand, we are using the `pick` instruction, in the definition of `genDvfOrders`, to choose the elements which are added to the vector field; however, this operator does not provide an actual method to select those elements.

4 An Effective Implementation: From Haskell to COQ

In the previous section, we have presented a non deterministic algorithm to construct an admissible discrete vector field from a matrix. Such an abstract version has been described on high-level datastructures; now, we are going to obtain from it a *refined* version, based on datastructures closer to machine representation, which will be executable.

The necessity of an executable algorithm which construct an admissible discrete vector field stems from the fact that they will play a key role to study biomedical images. There are several algorithms to construct an admissible discrete vector field; the one that we will use is explained in [24] (from now on, called RS's algorithm; RS stands for Romero-Sergeraert). The implementation of this algorithm will be executable but the proof of its correctness will be much more difficult than the one presented in the previous section.

4.1 The Romero-Sergeraert Algorithm

The underlying idea of the RS algorithm is that given an admissible discrete vector field, we try to enlarge it adding new vectors which preserve the admissibility property. We can define algorithmically the RS algorithm as follows.

Algorithm 11 (The RS Algorithm)

Input: a matrix M with coefficients in \mathbb{Z} .

Output: an admissible discrete vector field for M .

Description:

1. Initialize the vector field, V , to the void vector field.
2. Initialize the relations, $ords$, to nil.
3. For every row, i , of M :
 - 3.1. Search the first entry of the row equal to 1 or -1 , j .
 - 3.2. If (i, j) can be added to the vector field; that is, if we add it to V and generate all the relations, the properties of an admissible discrete vector field are preserved.
 - then:**
 - Add (i, j) to V .
 - Add to $ords$ the corresponding relations generated from (i, j) .
 - Go to the next row and repeat from Step 3.
 - else:** look for the next entry of the row whose value is 1 or -1 .
 - If there is not any.
 - then:** go to the next row and repeat from Step 3.
 - else:** go to Step 3.2 with j the column of the entry whose value is 1 or -1 .

In order to clarify how this algorithm works, let us construct an admissible discrete vector field from the following matrix.

$$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

We start with the void vector field $V = \{\}$. Running the successive rows, we find $M[1, 1] = 1$, and we include the vector $(1, 1)$ to V , obtaining $V = \{(1, 1)\}$. Then, let us add the relations that, in this case is $1 > 2$ because $M[2, 1] \neq 0$. So, it will be forbidden to incorporate the relation $2 > 1$ as the cycle $1 > 2 > 1$ would appear. Besides, the row 1 and the column 1 are now used and cannot be used anymore. So, we go on with the second row and find $M[2, 1] = 1$, but we cannot add $(2, 1)$ as we have just said. Moreover, the element $(2, 2)$ can not be incorporated because the cycle $1 > 2 > 1$ would be created. So, we continue and find the next element, $M[2, 3] = 1$. This does not create any cycle and satisfies the properties of a discrete vector field. Then, we obtain $V = \{(2, 3), (1, 1)\}$ and the relations $2 > 3$ and $2 > 4$. Running the next row, the first element equal to 1 is in the position $(3, 3)$, but we cannot include it due to the admissibility property. Therefore, we try with the last element of this row $(3, 4)$. No relation is generated in this case because in this column the only non null element is in the chosen position. So, $V = \{(3, 4), (2, 3), (1, 1)\}$. Finally, we run the last row. The elements that could be added are $(4, 2), (4, 3)$, but in both cases we would have to append the relation $4 > 2$. This would generate a cycle with one of the previous restrictions, $2 > 4$. So, we obtain $V = \{(3, 4), (2, 3), (1, 1)\}$ and the relations are: $1 > 2, 2 > 3$ and $2 > 4$.

In general, Algorithm [11](#) can be applied over matrices with coefficients in a general ring \mathcal{R} . From now on, we will work with $\mathcal{R} = \mathbb{Z}_2$, since this is the usual ring when working with monochromatic images in the context of Digital Algebraic Topology.

The development of a formally certified implementation of the RS algorithm has followed the methodology presented in [22](#). Firstly, we implement a version of our programs in *Haskell* [19](#), a *lazy* functional programming language. Subsequently, we intensively test our implementation using *QuickCheck* [5](#), a tool which allows one to automatically test properties about programs implemented in Haskell. Finally, we verify the correctness of our programs using the COQ *interactive* proof assistant and its SSREFLECT library.

4.2 A Haskell Program

The choice of Haskell to implement our programs was because both the code and the programming style is similar to the ones of COQ. In this programming language, we have defined the programs which implement the RS algorithm. The description of the main function is shown here:

gen_admdvf_ord M : From a matrix M with coefficients in \mathbb{Z}_2 , represented as a list of lists, this function generates an admissible discrete vector field for M , encoded by a list of natural pairs, and the relations, a list of lists of natural numbers.

Let us emphasize that the function `gen_admdvf_ord` returns a pair of elements. The former one, `(gen_admdvf_ord M).1`, is a discrete vector field and the latter one, `(gen_admdvf_ord M).2`, corresponds to the relations associated with the vector field. To provide a better understanding of these tools, let us apply them in the example presented in Subsection 4.1.

```
> gen_admdvf_ord [[1,1,0,0],[1,1,1,0],[0,0,1,1],[0,1,1,0]]
[[[(3,4),(2,3),(1,1)],[[2,4],[2,3],[1,2],[1,2,4],[1,2,3]]]]
```

Let us note that we return the transitive closure of the relations between the first components of the pairs of the discrete vector field. This will make the proof of the correctness of our programs easier.

4.3 Testing with QuickCheck

Using QuickCheck can be considered as a good starting point towards the formal verification of our programs. On the one hand, a specification of the properties which must be satisfied by our programs is given (a necessary step in the formalization process). On the other hand, before trying a formal verification of our programs (a quite difficult task) we are testing them, a process which can be useful in order to detect bugs.

In our case, we want to check that the output by `gen_admdvf_ord` gives us an admissible discrete vector field. Then, let M be a matrix over \mathbb{Z}_2 with m rows and n columns, $V = (a_i, b_i)_i$ be a discrete vector field from M and $ords$ be the transitive closure of the relations associated with V , the properties to test are the ones coming from Definition 10 and the admissibility property adapted to the \mathbb{Z}_2 case.

1. $1 \leq a_i \leq m$ and $1 \leq b_i \leq n$.
2. $\forall i, M(a_i, b_i) = 1$.
3. $(a_i)_i$ (resp. $(b_i)_i$) are pairwise different.
4. $ords$ does not have any loop (admissibility property).

These four properties has been encoded in Haskell by means of a function called `isAdmVecfield`. To test in QuickCheck that our implementation of the RS algorithm fulfills the specification given in `isAdmVecfield`, the following *property definition*, using QuickCheck terminology, is defined.

```
condAdmVecfield M =
let advf = (gen_admdvf_ord M) in isAdmVecfield M (advf.1) (advf.2)
```

The definition of `condAdmVecfield` states that given a matrix M , the output of `gen_admdvf_ord`, both the discrete vector field (first component) and the relations (second component) from M , fulfill the specification of the property called `isAdmVecfield`. Now, we can test whether `condAdmVecfield` satisfies such a property.

```
> quickCheck condAdmVecfield
+++ OK, passed 100 tests.
```

The result produced by QuickCheck when evaluating this statement, means that QuickCheck has generated 100 random values for M , checking that the property was true for all these cases. To be more precise, QuickCheck generates 100 random matrices over \mathbb{Z} ; however, our algorithm works with matrices over \mathbb{Z}_2 , then to overcome this pitfall we have defined a function which transforms matrices over \mathbb{Z} into matrices over \mathbb{Z}_2 turning even entries of the matrices into 0's and odd entries into 1's. In this way, we obtain random matrices over \mathbb{Z}_2 to test our programs.

4.4 Formalization in COQ /SSREFLECT

After testing our programs, and as a final step to confirm their correctness, we can undertake the challenge of formally verifying them.

First of all, we define the data types related to our programs, which are effective matrices, vector fields and relations. We have tried to keep them as close as possible to the Haskell ones; therefore, a matrix is represented by means of a list of lists over \mathbb{Z}_2 , a vector field is a sequence of natural pairs and finally, the relations is a list of lists of natural numbers.

Definition `Z2 := Fp_fieldType 2.`

Definition `matZ2 := seq (seq Z2).`

Definition `vectorfield := seq (prod nat nat).`

Definition `orders := seq (seq nat).`

Afterwards, we translate both the programs and the properties, which were specified during the testing of the programs, from Haskell to COQ, a task which is quite direct since these two systems are close.

Then, we have defined a function `isAdmVecfield` which receives as input a matrix over \mathbb{Z}_2 , a vector field and the relations and checks if the properties, explained in Subsection 4.3, are satisfied.

Definition `isAdmVecfield (M:matZ2)(vf:vectorfield)(ord:orders):=`
`((longmn (size M) (getfirstElemseq vf)) /\`
`(longmn (size (nth [::] M 0)) (getsndElemseq vf))) /\`
`(forall i j:nat, (i , j) \in vf -> compij i j M = 1) /\`
`((uniq (getfirstElemseq vf)) /\ (uniq (getsndElemseq vf))) /\`
`(admissible ord).`

Finally, we have proved the theorem `genDvfisVecfield` which says that given a matrix M , the output produced by `gen_admdvf_ord` satisfies the properties specified in `isAdmVecfield`.

Theorem `genDvfisVecfield (M:matZ2):`
`let advf := (gen_admdvf_ord M) in`
`isAdmVecfield M (advf.1) (advf.2).`

We have split the proof of the above theorem into 4 lemmas which correspond with each one of the properties that should be fulfilled to have an admissible discrete vector field. For instance, the lemma associated with the first property of the definition of a discrete vector field is the following one.

```
Lemma propSizef (M:matZ2):
  let advf := (gen_admdvf_ord M).1 in
  (longmn (size M) (getfirstElemseq advf) /\
   (longmn (size (nth nil M 0))(getsndElemseq advf)).
```

Both the functions which implement the RS algorithm and the ones which specify the definitional properties of admissible discrete vector fields are defined using a *functional style*; that is, our programs are defined using *pattern-matching* and *recursion*. Therefore, in order to reason about our recursive functions, we need elimination principles which are fitted for them. To this aim, we use the tool presented in [3] which allows one to reason about complex recursive definitions since COQ does not directly generate elimination principles for complex recursive functions. Let us see how the tool presented in [3] works.

In our development of the implementation of the RS algorithm, we have defined a function, called `subm`, which takes as arguments a natural number, `n`, and a matrix, `M`, and removes the first `n` rows of `M`. The inductive scheme associated with `subm` is set as follows.

```
Functional Scheme subm_ind := Induction for subm Sort Prop.
```

Then, when we need to reason about `subm`, we can apply this scheme with the corresponding parameters using the instruction `functional induction`. However, as we have previously said both our programs to define the RS algorithm and the ones which specify the properties to prove are recursive. Then, in several cases, it is necessary to merge several inductive schemes to induction simultaneously on several variables. For instance, let M be a matrix and M' be a submatrix of M where we have removed the $(k-1)$ first rows of M ; then, we want to prove that $\forall j, M(i, j) = M'(i - k + 1, j)$. This can be stated in COQ as follows.

```
Lemma Mij_subM (i k: nat) (M: matZ2):
  k <= i -> k != 0 -> let M' := (subm k M) in
  M i j == M' (i - k + 1) j.
```

To prove this lemma it is necessary to induct simultaneously on the parameters `i`, `k` and `M`, but the inductive scheme generated from `subm` only applies induction on `k` and `M`. Therefore, we have to define a new recursive function, called `Mij_subM_rec`, to provide a proper inductive scheme to prove this theorem.

```
Fixpoint Mij_subM_rec (i k: nat) (M: matZ2) :=
match k with
|0 => M
|S p => match M with
|nil => nil
|hM::tM => if (k == 1)
```

```

      then a::b
    else (Mij_subM_rec p (i- 1) tM)
  end
end.

```

This style of proving functional programs in COQ is the one followed in the development of the proof of Theorem `genDvfnisVecfield`.

4.5 Experimental Results

Using the same methodology presented throughout this section, we are working in the formalization of the algorithm which, from a matrix and an admissible discrete vector field on it, produces a reduced matrix preserving the homological properties of the original one. Up to now, we have achieved a Haskell implementation which has been both tested with QuickCheck and translated into COQ; however, the proof of its correctness remains as further work.

Anyway, as we have a COQ implementation of that procedure, we can execute some examples inside this proof assistant. Namely, we have integrated the programs presented in this paper with the ones devoted to the computation of homology groups of digital images introduced in [15]. The reason to carry out the computations within COQ, instead of transferring the verified algorithms into a more efficient programming language like OCaml or Haskell, is the chance that this approach provides us to reuse the result of our homological computations for further proofs. It is worth noting that the outputs produced by external programs are untrusted so they cannot be imported inside COQ.

As a benchmark we have considered matrices coming from 500 randomly generated images. The size of the matrices associated with those images was initially around 100×300 , and after the reduction process the average size was 5×50 . Using the original matrices COQ takes around 12 seconds to compute the homology from the matrices; on the contrary, using the reduced matrices COQ only needs milliseconds. Furthermore, as we will see in the following section, we have studied some images which are originated from a real biomedical problem.

5 Application to Biomedical Images

Biomedical images are a suitable benchmark for testing our programs. On the one hand, the amount of information included in this kind of images is usually quite big; then, a process able to reduce those images but keeping the homological properties can be really useful. On the other hand, software systems dealing with biomedical images must be trustworthy; this is our case since we have formally verified the correctness of our programs.

As an example, we can consider the problem of counting the number of *synapses* in a neuron. Synapses [4] are the points of connection between neurons and are related to the computational capabilities of the brain. Therefore, the treatment of neurological diseases, such as Alzheimer, may take advantage of procedures modifying the number of synapses [8].

Up to now, the study of the *synaptic density evolution* of neurons was a time-consuming task since it was performed, mainly, manually. To overcome this issue, an automatic method was presented in [16]. Briefly speaking, such a process can be split into two parts. Firstly, from three images of a neuron (the neuron with two antibody markers and the structure of the neuron), a monochromatic image is obtained, see Figure 1. In such an image, each connected component represents a synapse. So, the problem of measuring the number of synapses is translated into a question of counting the connected components of a monochromatic image.

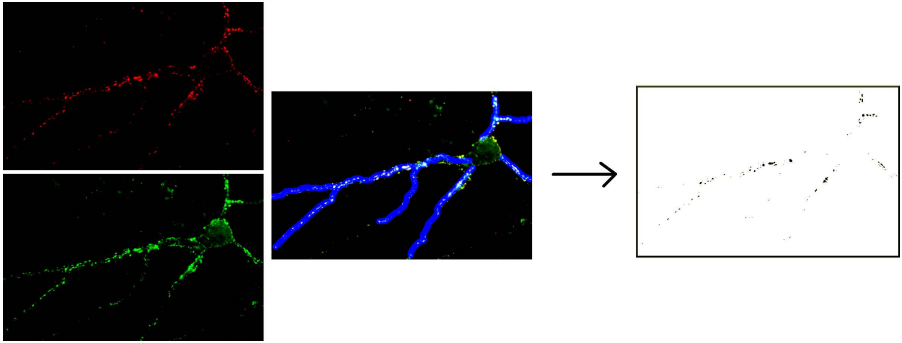


Fig. 1. Synapses extraction from three images of a neuron

In the context of Algebraic Digital Topology, this issue can be tackled by means of the computation of the homology group H_0 of the monochromatic image. This task can be performed in COQ through the formally verified programs which were presented in [15]. Nevertheless, such programs are not able to handle images like the one of the right side of Figure 1 due to its size (let us remark that COQ is a proof assistant tool and not a computer algebra system). In order to overcome this drawback, as we have explained at the end of the previous section, we have integrate our reduction programs with the ones presented in [15]. Using this approach, we can successfully compute the homology of the biomedical images in just 25 seconds, a remarkable time for an execution inside COQ.

6 Conclusions and Further Work

In this paper, we have given the first step towards the formal verification of a procedure which allows one to study homological properties of big digital images inside COQ. The underlying idea consists in building an admissible discrete vector field on the matrices associated with an image and, subsequently reduce those matrices but preserving the homology.

² The same images with higher resolution can be seen in <http://www.unirioja.es/cu/joheras/synapses/>

Up to now, we have certified the former step of this procedure, the construction of an admissible discrete vector field from a matrix, both in an abstract and a concrete way. The reason because the abstract formalization is useful is twofold: on the one hand, it provides a high-level theory close to usual mathematics, and, on the other hand, it has been refined to obtain the effective construction of admissible discrete vector fields. As we have explained, there are several heuristics to construct an admissible discrete vector field from a matrix, the one that we have chosen is the RS algorithm [24] which produces, as we have experimentally seen, quite large discrete vector fields, a desirable property for these objects. The latter step, the process to reduce the matrices, is already specified in COQ, but the proof of its correctness is still an ongoing work.

The suitability of our approach has been tested with several examples coming from randomly generated images and also real images associated with a biomedical problem, the study of synaptic density evolution. The results which have been obtained are remarkable since the amount of time necessary to compute homology groups of such images inside COQ is considerably reduced (in fact, it was impossible in the case of biomedical images).

As further work, we have to deal with some formalization issues. Namely, we have to verify that a reduction can be constructed from a matrix and an admissible discrete vector field to a reduced matrix. Moreover, we have hitherto worked with matrices over the ring \mathbb{Z}_2 ; the more general case of matrices with coefficients in a ring \mathcal{R} (with convenient constructive properties) should be studied.

As we have seen in Subsection 4.4, it is necessary the definition of inductive schema which fits to our complex recursive programs. Then, this opens the door to an integration between COQ and the ACL2 Theorem Prover [20]. ACL2 has good heuristics to generate inductive schemes from recursive functions; so, we could translate our functional programs from COQ to ACL2, generate the inductive schemes in ACL2; and finally return such inductive schemes to COQ. Some preliminary experiments have been performed to automate that process, obtaining encouraging results.

In a different research line, we can consider the study of more complex biomedical problems using our certified programs. As an example, the recognition of the structure of neurons seems to involve the computation of homology groups in higher dimensions; a question which could be tackled with our tools.

References

1. Mathematical components team homepage, <http://www.msr-inria.inria.fr/Projects/math-components>
2. Aransay, J., Ballarin, C., Rubio, J.: A mechanized proof of the Basic Perturbation Lemma. *Journal of Automated Reasoning* 40(4), 271–292 (2008)
3. Barthe, G., Courtieu, P.: Efficient Reasoning about Executable Specifications in Coq. In: Carreño, V.A., Muñoz, C.A., Tahar, S. (eds.) TPHOLs 2002. LNCS, vol. 2410, pp. 31–46. Springer, Heidelberg (2002)
4. Bear, M., Connors, B., Paradiso, M.: *Neuroscience: Exploring the Brain*. Lippincott Williams & Wilkins (2006)

5. Claessen, K., Hughes, J.: QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In: ACM SIGPLAN Notices, pp. 268–279. ACM Press (2000)
6. Cohen, C., Mahboubi, A.: Formal proofs in real algebraic geometry: from ordered fields to quantifier elimination (2011), <http://hal.inria.fr/inria-00593738>
7. COQ development team. The COQ Proof Assistant, version 8.3. Technical report (2010)
8. Cuesto, G., et al.: Phosphoinositide-3-Kinase Activation Controls Synaptogenesis and Spinogenesis in Hippocampal Neurons. *The Journal of Neuroscience* 31(8), 2721–2733 (2011)
9. Domínguez, C., Rubio, J.: Effective Homology of Bicomplexes, formalized in Coq. *Theoretical Computer Science* 412, 962–970 (2011)
10. Dousson, X., Rubio, J., Sergeraert, F., Siret, Y.: The Kenzo program. Institut Fourier, Grenoble (1998), <http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/>
11. Forman, R.: Morse theory for cell complexes. *Advances in Mathematics* 134, 90–145 (1998)
12. Gonthier, G.: Formal proof - The Four-Color Theorem, vol. 55. Notices of the American Mathematical Society (2008)
13. Gonthier, G., Mahboubi, A.: An introduction to small scale reflection in Coq. *Journal of Formal Reasoning* 3(2), 95–152 (2010)
14. Graham, P.: ANSI Common Lisp. Prentice Hall (1996)
15. Heras, J., Dénès, M., Mata, G., Mörtberg, A., Poza, M., Siles, V.: Towards a certified computation of homology groups for digital images. In: Proceedings 4th International Workshop on Computational Topology in Image Context (CTIC 2012). LNCS (to appear, 2012)
16. Heras, J., Mata, G., Poza, M., Rubio, J.: Homological processing of biomedical digital images: automation and certification. In: 17th International Conferences on Applications of Computer Algebra. Computer Algebra in Algebraic Topology and its Applications Session (2011)
17. Heras, J., Pascual, V., Rubio, J.: Proving with ACL2 the Correctness of Simplicial Sets in the Kenzo System. In: Alpuente, M. (ed.) LOPSTR 2010. LNCS, vol. 6564, pp. 37–51. Springer, Heidelberg (2011)
18. Jacobson, N.: Basic Algebra II, 2nd edn. W. H. Freeman and Company (1989)
19. Jones, S.P., et al.: The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming* 13(1), 0–255 (2003), <http://www.haskell.org>
20. Kaufmann, M., Moore, J.S.: ACL2 version 4.3 (2011)
21. Lambán, L., Martín-Mateos, F.J., Rubio, J., Ruiz-Reina, J.L.: Applying ACL2 to the Formalization of Algebraic Topology: Simplicial Polynomials. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 200–215. Springer, Heidelberg (2011)
22. Mörtberg, A.: Constructive algebra in functional programming and type theory. In: *Mathematics, Algorithms and Proofs 2010* (2010), <http://wiki.portal.chalmers.se/cse/pmwiki.php/ForMath/PapersAndSlides>
23. Romero, A., Rubio, J.: Homotopy groups of suspended classifying spaces: an experimental approach. To be published in *Mathematics of Computation* (2012)
24. Romero, A., Sergeraert, F.: Discrete Vector Fields and Fundamental Algebraic Topology (2010), <http://arxiv.org/abs/1005.5685v1>
25. Rubio, J., Sergeraert, F.: Constructive Algebraic Topology. *Bulletin des Sciences Mathématiques* 126(5), 389–412 (2002)

Towards the Formal Specification and Verification of Maple Programs*

Muhammad Taimoor Khan¹ and Wolfgang Schreiner²

¹ Doktoratskolleg Computational Mathematics

² Research Institute for Symbolic Computation

Johannes Kepler University

Linz, Austria

muhammad.khan@dk-compmath.jku.at,

Wolfgang.Schreiner@risc.jku.at

<http://www.risc.jku.at/people/mtkhan/dk10/>

Abstract. In this paper, we present our ongoing work and initial results on the formal specification and verification of *MiniMaple* (a substantial subset of Maple with slight extensions) programs. The main goal of our work is to find behavioral errors in such programs w.r.t. their specifications by static analysis. This task is more complex for widely used computer algebra languages like Maple as these are fundamentally different from classical languages: they support non-standard types of objects such as symbols, unevaluated expressions and polynomials and require abstract computer algebraic concepts and objects such as rings and orderings etc. As a starting point we have defined and formalized a syntax, semantics, type system and specification language for *MiniMaple*.

1 Introduction

Computer algebra programs written in symbolic computation languages such as Maple and Mathematica sometimes do not behave as expected, e.g. by triggering runtime errors or delivering wrong results. There has been a lot of research on applying formal techniques to classical programming languages, e.g. Java [10], C# [1] and C [2]; we aim to apply similar techniques to computer algebra languages, i.e. to design and develop a tool for the static analysis of computer algebra programs. This tool will find errors in programs annotated with extra information such as variable types and method contracts, in particular type inconsistencies and violations of methods preconditions.

As a starting point, we have defined the syntax and semantics of a subset of the computer algebra language Maple called *MiniMaple*. Since type safety is a prerequisite of program correctness, we have formalized a type system for *MiniMaple* and implemented a corresponding type checker. The type checker has been applied to the Maple package *DifferenceDifferential* [7] developed at

* The research was funded by the Austrian Science Fund (FWF): W1214-N15, project DK10.

our institute for the computation of bivariate difference-differential dimension polynomials. Furthermore, we have defined a specification language to formally specify the behavior of *MiniMaple* programs. As the next step, we will develop a verification calculus for *MiniMaple* respectively a corresponding tool to automatically detect errors in *MiniMaple* programs with respect to their specifications. An example-based short demonstration of this work is presented in the accompanying paper [16].

Figure 1 gives a sketch of the envisioned system (the verifier component is under development); any *MiniMaple* program is parsed to generate an abstract syntax tree (AST). The AST is then annotated by type information and used by the verifier to check the correctness of a program. Error and information messages are generated by the respective components.

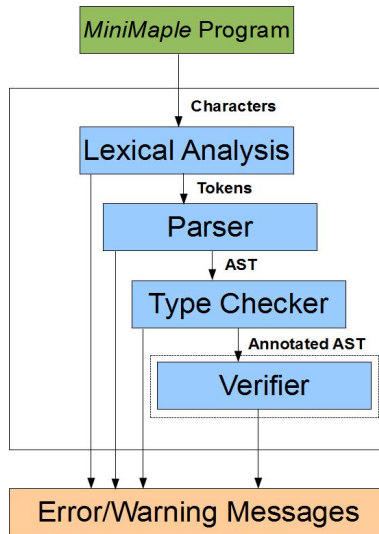


Fig. 1. Sketch of the System

There are various computer algebra languages, Mathematica and Maple being the most widely used by far, both of which are dynamically typed. We have chosen for our work Maple because of its simpler, more classical and imperative structure. Still we expect that the results we derive for type checking respective formal specification Maple can be applied to Mathematica, as Mathematica shares with Maple many concepts such as basic kinds of runtime objects.

During our study, we found the following special features for type checking respective formal specification of Maple programs (which are typical for most computer algebra languages):

- The language supports some non-standard types of objects, e.g. symbols, unevaluated expressions and polynomials.
- The language uses type information to direct the flow of control in the program, i.e. it allows some runtime type-checks which selects the respective code-block for further execution.
- The language lacks in the use of abstract data types, which are necessary for the adequate specification of computer algebra functions.

The rest of the paper is organized as follows: in Section 2, we describe state of the art related to our work. In Section 3, we introduce the syntax of *MiniMaple* by an example. In Section 4, we briefly explain our type system for *MiniMaple*. In Section 5, we discuss our formal specification language for *MiniMaple*. In Section 6, we highlight the interesting features of a formal semantics of *MiniMaple*. Section 7 presents conclusions and future work.

2 State of the Art

In this section we first sketch state of the art of type systems for Maple and then discuss the application of formal techniques to computer algebra languages.

Although there is no complete static type system for Maple; there have been several approaches to exploit the type information in Maple for various purposes. For instance, the Maple package Gauss [17] introduced parameterized types in Maple. Gauss ran on top of Maple and allowed to implement generic algorithms in Maple in an AXIOM-like manner. The system supported parameterized types and parameterized abstract types, however these were only checked at runtime. The package was introduced in Maple V Release 2 and later evolved into the *domains* package. In [6], partial evaluation is applied to Maple. The focus of the work is to exploit the available type information for generating specialized programs from generic Maple programs. The language of the partial evaluator has similar syntactic constructs (but fewer expressions) as *MiniMaple* and supports very limited types e.g. integers, rationals, floats and strings. The problem of statically type-checking *MiniMaple* programs is related to the problem of statically type-checking scripting languages such as Ruby [13], but there are also fundamental differences due to the different language paradigms.

In comparison to the approaches discussed above, *MiniMaple* uses the type annotations provided by Maple for static analysis. It supports a substantial subset of Maple types in addition to named types.

Various specification languages have been defined to formally specify the behavior of programs written in standard classical programming languages, e.g. Java Modeling Language (JML) [10] for Java, Spec# [1] for C# and ACSL [2] for ANSI C: these specification languages are used by various tools for extended static checking and verification [8] of programs written in the corresponding languages. Also variously the application of formal methods to computer algebra has been investigated. For example [9] applied the formal specification language Larch [11] to the computer algebra system AXIOM respective its programming

language Aldor. A methodology for Aldor program analysis and verification was devised by defining abstract specifications for AXIOM primitives and then providing an interface between these specifications and Aldor code. The project FoCaLiZe [20] aims to provide a programming environment for computer algebra to develop certified programs to achieve high levels of software security. The environment is based on functional programming language FoCal, which also supports some object-oriented features and allows the programmer to write formal specifications and proofs of programs. The work presented in [5] aims at finding a mathematical description of the interfaces between Maple routines. The paper mainly presents the study of the actual contracts in use by Maple routines. The contracts are statements with certain (static and dynamic) logical properties. The work focused to collect requirements for the pure type inference engine for existing Maple routines. The work was extended to develop the partial evaluator for Maple mentioned above [6].

The specification language for *MiniMaple* fundamentally differs from those for classical languages such that it supports some non-standard types of objects, e.g. symbols, unevaluated expressions and polynomials etc. The language also supports abstract data types to formalize abstract mathematical concepts, while the existing specification languages are weaker in such specifications. In contrast to the computer algebra specification languages above, our specification language is defined for the commercially supported language Maple, which is widely used but was not designed to support static analysis (type checking respectively verification). The challenge here is to overcome those particularities of the language that hinder static analysis.

3 *MiniMaple*

MiniMaple is a simple but substantial subset of Maple that covers all the syntactic domains of Maple but has fewer alternatives in each domain than Maple; in particular, Maple has many expressions which are not supported in our language. The complete syntactic definition of *MiniMaple* is given in [14]. The grammar of *MiniMaple* has been formally specified in BNF from which a parser for the language has been automatically generated with the help of the parser generator ANTLR.

The top level syntax for *MiniMaple* is as follows:

$$\begin{aligned} \textit{Prog} &:= \textit{Cseq}; \\ \textit{Cseq} &:= \textit{EMPTY} \mid \textit{C}, \textit{Cseq} \\ \textit{C} &:= \dots \mid \textit{I}, \textit{Iseq} := \textit{E}, \textit{Eseq} \mid \dots \end{aligned}$$

A program is a sequence of commands, there is no separation between declaration and assignment.

```

1. status:=0;
2. prod := proc(l::list(Or(integer,float)):[integer,float];
3.     global status;
4.     local i::integer, x::Or(integer,float), si::integer:=1, sf::float:=1.0;
5.     for i from 1 by 1 to nops(l) while (running) do
6.         x:=l[i];
7.         if type(x,integer) then
8.             if (x = 0) then
9.                 return [si,sf];
10.            else
11.                si:=si*x;
12.            end if;
13.        elif type(x,float) then
14.            if (x < 0.5) then
15.                return [si,sf];
16.            else
17.                sf:=sf*x;
18.            end if;
19.        end if;
20.    end do;
21.    return [si,sf];
22. end proc;

```

Listing 1. An example *MiniMaple* program

Listing 1 gives an example of a *MiniMaple* program which we will use in the following sections for the discussion of type checking and behavioral specification. The program consists of an assignment initializing a global variable *status* and an assignment defining a procedure *prod* followed by the application of the procedure. The procedure takes a list of integers and floats and computes the product of these integers and floats separately; it returns as a result a tuple of the products. The procedure may also terminate prematurely for certain inputs, i.e. either for an integer value 0 or for a float value less than 0.5 in the list; in this case the procedure computes the respective products just before the index at which the aforementioned terminating input occurs.

As one can see from the example, we make use of the type annotations that Maple introduced for runtime type checking. In particular, we demand that function parameters, function results and local variables are correspondingly type annotated. Based on these annotations, we define a language of types and a corresponding type system for the static type checking of *MiniMaple* programs.

4 A Type System for *MiniMaple*

A *type* is (an upper bound on) the range of values of a variable. A *type system* is a set of formal typing rules to determine the variables types from the

text of a program. A type system prevents *forbidden errors* during the execution of a program. It completely prevents the *untrapped errors* and also a large class of *trapped errors*. *Untrapped errors* may go unnoticed for a while and later cause an arbitrary behavior during execution of a program, while *trapped errors* immediately stop execution [4].

A type system in essence is a decidable logic with various kinds of *judgments*; for example the typing judgment

$$\pi \vdash E:(\tau)\mathbf{exp}$$

can be read as “in the given type environment π , E is a well-typed expression of type τ ”. A type system is *sound*, if the deduced types indeed capture the program values exhibited at runtime.

In the following we describe the main properties of a type system for *MiniMaple*. Subsection 4.1 sketches its design and Subsection 4.2 presents its implementation and application. A proof of the soundness of the type system remains to be performed.

```

1. status:=0;
2. prod := proc(l:list(Or(integer,float))):[integer,float];
3.     #  $\pi=\{l:\text{list}(\text{Or}(\text{integer},\text{float}))\}$ 
4.     global status;
5.     local i, x::Or(integer,float), si::integer:=1, sf::float:=1.0;
6.     #  $\pi=\{\dots, i:\text{symbol}, x:\text{Or}(\text{integer},\text{float}),\dots, \text{status}:\text{anything}\}$ 
7.     for i from 1 by 1 to nops(l) do
8.         x:=l[i];
9.         status:=i;
10.        #  $\pi=\{\dots, i:\text{integer}, \dots, \text{status}:\text{integer}\}$ 
11.        if type(x,integer) then
12.            #  $\pi=\{\dots, i:\text{integer}, x:\text{integer}, si:\text{integer}, \dots, \text{status}:\text{integer}\}$ 
13.            if (x = 0) then
14.                return [si,sf];
15.            end if;
16.            si:=si*x;
17.        elif type(x,float) then
18.            #  $\pi=\{\dots, i:\text{integer}, x:\text{float}, \dots, sf:\text{float}, \text{status}:\text{integer}\}$ 
19.            if (x < 0.5) then
20.                return [si,sf];
21.            end if;
22.            sf:=sf*x;
23.        end if;
24.        #  $\pi=\{\dots, i:\text{integer}, x:\text{Or}(\text{integer},\text{float}),\dots, \text{status}:\text{integer}\}$ 
25.    end do;
26.    #  $\pi=\{\dots, i:\text{symbol}, x:\text{Or}(\text{integer},\text{float}),\dots, \text{status}:\text{anything}\}$ 
27.    status:=1;
28.    #  $\pi=\{\dots, i:\text{symbol}, x:\text{Or}(\text{integer},\text{float}),\dots, \text{status}:\text{integer}\}$ 
29.    return [si,sf];
30. end proc;
31. result := prod([1, 8.54, 34.4, 6, 8.1, 10, 12, 5.4]);

```

Listing 2. A *MiniMaple* procedure type-checked

4.1 Design

MiniMaple uses Maple type annotations for static type checking, which gives rise to the following language of types:

$$T ::= \mathbf{integer} \mid \mathbf{boolean} \mid \mathbf{string} \mid \mathbf{float} \mid \mathbf{rational} \mid \mathbf{anything} \\ \mid \{ T \} \mid \mathbf{list}(T) \mid [Tseq] \mid \mathbf{procedure}[T](Tseq) \\ \mid I(Tseq) \mid \mathbf{Or}(Tseq) \mid \mathbf{symbol} \mid \mathbf{void} \mid \mathbf{uneval} \mid I$$

The language supports the usual concrete data types, sets of values of type T ($\{ T \}$), lists of values of type T ($\mathbf{list}(T)$) and records whose members have the values of types denoted by a type sequence $Tseq$ ($[Tseq]$). Type **anything** is the super-type of all types. Type **Or**($Tseq$) denotes the union type of various types, type **uneval** denotes the values of unevaluated expressions, e.g. polynomials, and type **symbol** is a name that stands for itself if no value has been assigned to it. User-defined data types are referred by I while $I(Tseq)$ denotes tuples (of values of types $Tseq$) tagged by a name I .

A sub-typing relation ($<$) is defined among types, i.e. **integer** $<$ **rational** $<$... $<$ **anything**, such that **integer** is a sub-type of **rational** and the type **anything** is the super-type of all types.

In the following, we demonstrate the problems arising from type checking *MiniMaple* programs using the example presented in the previous section.

Global Variables. Global variables (declarations) can not be type annotated; therefore to global variables values of arbitrary types can be assigned in Maple. We introduce *global* and *local* contexts to handle the different semantics of the variables inside and outside of the body of a procedure i.e. respective loop.

- In a *global* context new variables may be introduced by assignments and the types of variables may change arbitrarily by assignments.
- In a *local* context variables can only be introduced by declarations. The types of variables can only be *specialized* i.e. the new value of a variable should be a sub-type of the declared variable type. The sub-typing relation is observed while specializing the types of variables.

Type Tests. A predicate $\mathbf{type}(E, T)$ (which is true if the value of expression E has type T) may direct the control flow of a program. If this predicate is used in a conditional, then different branches of the conditional may have different type information for the same variable. We keep track of the type information introduced by the different type tests from different branches to adequately reason about the possible types of a variable. For instance, if a variable x has type $\mathbf{Or}(\mathbf{integer}, \mathbf{float})$, in a conditional statement where the "then" branch is guarded by a test $\mathbf{type}(x, \mathbf{integer})$, in the "else" branch x has automatically type \mathbf{float} . A warning is generated, if a test is redundant (always yields true or false).

For our example program our type system will generate the type information as depicted in Listing 2. The program is annotated with the type environment (a partial function from identifiers to their corresponding types) of the form $\# \pi = \{variable: type, \dots\}$. For example, the type environment at line 6 shows the types of the respective variables as determined by the static analysis of parameter and identifier declarations (**global** and **local**).

The static analysis of the two branches of the conditional command in the body of the loop introduces the type environments at lines 12 and 18 respectively; the type of variable x is determined as **integer** and **float** by the conditional type-expressions respectively.

There is more type information to direct the program control flow for an identifier x introduced by an expression **type**(E, T) at lines 11 and 17.

By analyzing the conditional command as a whole, the type of variable x is determined as **Or(integer, float)** (at line 24), i.e. the union type of the two types determined by the respective branches.

The local type information introduced/modified by the analysis of body of loop does not effect the global type information. The type environment at lines 6 and 26 reflects this fact for variables $status$, i and x . This is because of the fact that the number of loop iterations might have an effect on the type of the variable otherwise and one cannot determine the concrete type by the static analysis. To handle this non-determination of types we put a reasonable upper bound (fixed point) on the types of such variables, namely the type of a variable prior to the body of a loop.

4.2 Formalization

In this subsection we explain the typing judgments and typing rules for some expressions and commands of *MiniMaple*. These judgments use the following kinds of objects (“Identifier” and ”Type“ are the syntactic domains of identifiers/variables and types of *MiniMaple* respectively):

- π : Identifier \rightarrow Type: a type environment, i.e. a (partial) function from identifiers to types.
- $c \in \{\text{global, local}\}$: a tag representing the context to check if the corresponding syntactic phrase is type checked inside/outside of the procedure/loop.
- $asgnset \subseteq$ Identifier: a set of assignable identifiers introduced by type checking the declarations.
- $\epsilon set \subseteq$ Identifier: a set of thrown exceptions introduced by type checking the corresponding syntactic phrase.
- $\tau set \subseteq$ Type: a set of return types introduced by type checking the corresponding syntactic phrase.
- $rflag \in \{\text{aret, not_aret}\}$: a return flag to check if the last statement of every execution of the corresponding syntactic phrase is a *return* command.

MiniMaple supports various types of expressions but boolean expressions are treated specially because of the test **type**(I, T) that gives additional type information about the expression. The typing judgment for boolean expressions

$$\pi \vdash E:(\pi_1)\mathbf{boolexp}$$

can be read as "with the given π , E is a well-typed boolean expression with new type environment π_1 ". The new type environment is produced as a fact of type test that might introduce new type information for an identifier.

The typing judgment for commands

$$\pi, c, \mathit{asnset} \vdash C:(\pi_1, \tau_{set}, \epsilon_{set}, rflag)\mathbf{comm}$$

can be read as "in the given type environment π , context c and an assignable set of identifiers asnset , C is a well-typed command and produces $(\pi_1, \tau_{set}, \epsilon_{set}, rflag)$ as type information". In the following we explain some typing rules to derive typing judgments for boolean expressions and conditional commands. These typing rules use different kinds of auxiliary functions and predicates as given below.

Auxiliary Functions

- *specialize*(π_1, π_2): specializes the identifiers of former type environment to the identifiers in the latter type environment w.r.t. their types.
- *combine*(π_1, π_2): combines the identifiers in the two environments with respect to their types.
- *superType*(τ_1, τ_2): returns the super-type between the two given types.

Auxiliary Predicates

- *canSpecialize*(π_1, π_2): returns true if all the common identifiers (in both type environments) have a super-type between their corresponding types.
- *superType*(τ_1, τ_2): returns true (in most cases) if the former type is general (super) type than the latter type. **Anything** is the super-type of all types.

Typing Rules. The typing rule for boolean expressions is as follows:

- **type**(I, T)

$$\frac{\pi \vdash I:(\tau_1)\mathbf{id} \quad \pi \vdash T:(\tau_2)\mathbf{type} \quad \mathit{superType}(\tau_1, \tau_2)}{\pi \vdash \mathbf{type}(I, T):(\{\tau_1, \tau_2\})\mathbf{boolexp}}$$

The phrase "**type**(I, T)" is a well-typed boolean expression if the declared type of identifier (τ_1) is the super-type of T (τ). The boolean expression may introduce new type information for the identifier.

The typing rule for the conditional command is given below:

– **if** E **then** $Cseq$ $Elif$ **end if**

$$\frac{\begin{array}{l} \pi \vdash E: (\pi')\mathbf{boolexp} \quad canSpecialize(\pi, \pi') \\ specialize(\pi, \pi'), c, asgnset \vdash Cseq: (\pi_1, \tau set_1, \epsilon set_1, rflag_1)\mathbf{cseq} \\ \pi, c, asgnset \vdash Elif: (\pi_2, \pi set, \tau set_2, \epsilon set_2, rflag_2)\mathbf{elif} \end{array}}{\pi, c, asgnset \vdash \mathbf{if} E \mathbf{then} Cseq Elif \mathbf{end}} \\ \mathbf{if}: (combine(\pi_1, \pi_2), \tau set_1 \cup \tau set_2, \epsilon set_1 \cup \epsilon set_2, ret(rflag_1, rflag_2))\mathbf{comm}$$

The phrase “**if** E **then** $Cseq$ $Elif$ **end if**“ is a well typed conditional command if the type of expression E does not conflict global type information. The conditional command combines the type environment of its two conditional branches (*if* and *elif*), because we are not sure which of the branches will be executed at runtime.

4.3 Application

Based on the type system sketched above we have implemented a type checker for *MiniMaple* [14] in Java (150+ classes and 15K+ lines of code). The type checker also handles the specification language of *MiniMaple*.

Figure 2 shows that the output of the type checker applied to a file containing the source code of the example program from the previous section. It shows that the file has successfully parsed and also presents the type annotations for the first assignment command. In the second part, it shows the resulting type environment with the associated program identifiers and their respective types introduced while type checking. The last message indicates that the program type checked correctly.

```
/home/taimoor/antlr3/Test6.m parsed with no errors.
Generating Annotated AST...
...
*****COMMAND-SEQUENCE-ANNOTATION START*****
PI -> [
prod:procedure[[integer,float]](list(Or(integer,float)))
status:integer
result:[integer,float]
]
RetTypeSet -> {}
ThrownExceptionSet -> {}
RetFlag -> not_aret
*****COMMAND-SEQUENCE-ANNOTATION END*****
Annotated AST generated.
The program type-checked correctly.
```

Fig. 2. Parsing and Type Checking the Program

The main test case for our type checker is the Maple package *Difference-Differential* [7] developed by Christian Dönch at our institute. The package provides algorithms for computing difference-differential dimension polynomials by

relative Gröbner bases in difference-differential modules according to the method developed by M. Zhou and F. Winkler [22].

We manually translated this package into a *MiniMaple* package so that the type checker can be applied. This translation consists of

- adding required type annotations and
- translating those parts of the package that are not directly supported into logically equivalent *MiniMaple* constructs.

No crucial typing errors have been found but some bad code parts have been identified that can cause problems, e.g., variables that are declared but not used (and therefore cannot be type checked) and variables that have duplicate global and local declarations.

5 A Formal Specification Language for *MiniMaple*

Based on the type system presented in the previous section, we have developed a formal specification language for *MiniMaple*. This language is a logical formula language which is based on Maple notations but extended by new concepts. The formula language supports various forms of quantifiers, logical quantifiers (**exists** and **forall**), numerical quantifiers (**add**, **mul**, **min** and **max**) and sequential quantifier (**seq**) representing truth values, numeric values and sequence of values respectively. We have extended the corresponding Maple syntax, e.g., logical quantifiers use typed variables and numerical quantifiers are equipped with logical conditions that filter values from the specified variable range. The example for these quantifiers is explained later in the procedure specification of this section. The use of this specification language is described in the conclusions.

Also the language allows to formally specify the behavior of procedures by pre- and post-conditions and other constraints; it also supports loop specifications and assertions. In contrast to specification languages such as JML, abstract data types can be introduced to specify abstract concepts and notions from computer algebra.

At the top of *MiniMaple* program one can declare respectively define mathematical functions, user-defined named and abstract data types and axioms. The syntax of specification declarations

$$\begin{aligned} \text{decl} ::= & \text{EMPTY} \\ & | (\text{define}(I, \text{rules}); \\ & | \text{'type}/I':=T; | \text{'type}/I'; \\ & | \text{assume}(\text{spec-expr});) \text{ decl} \end{aligned}$$

is mainly borrowed from Maple. The phrase “**define**(I, rules);” can be used for defining mathematical functions as shown in the following the factorial function:

$$\text{define}(\text{fac}, \text{fac}(0) = 1, \text{fac}(n::\text{integer}) = n * \text{fac}(n - 1));$$

User-defined data types can be declared with the phrase “**type**/ I := T ;” as shown in the following declaration of “ListInt” as the list of integers:

```
'type/ListInt':=list(integer);
```

The phrase "**'type/I';**" can be used to declare abstract data type with the name I , e.g. the following example shows the declaration of abstract data type "difference differential operator (DDO)".

```
'type/DDO';
```

The task of formally specifying mathematical concepts using abstract data types is more simpler as compared to their underlying representation with concrete data types. Also other related facts and the access functions of abstract concept can be formalized for better reasoning.

Axioms can be introduced by the phrase "**assume(spec-expr);**" as the following example shows an axiom that an operator is a difference-differential operator, if its each term is a difference-differential term, where d is an operator:

```
assume(isDDO(d) equivalent forall(i::integer, 1<=i and i<=terms(d) implies  

isDDOTerm(getTerm(d,i,1),getTerm(d,i,2),  

getTerm(d,i,3),getTerm(d,i,4));
```

Any predicate declaration can be introduced by the phrase "**I(spec-expr);**" as the following example shows a predicate that when a given field is supported:

```
inField(c);
```

The entities introduced by the specification declarations can be used in the following specifications.

A procedure specification consists of a pre-condition, the set of global variables that can be modified and the post condition, describing the relationship between pre and post state. By an optional exception clause we can specify the exceptional behavior of a procedure. The procedure specification syntax is influenced by the Java Modeling Language:

```
proc-spec ::= requires spec-expr;  

global Iseq;  

ensures spec-expr; excep-clause
```

Listing 3 shows an example for the procedure specification. The specification is a big logical disjunction to formulate two possible behaviors of the procedure:

1. when the procedure terminates normally and
2. when the procedure terminates prematurely.

```
(*@
```

```
requires true;  

global status;  

ensures  

  (status = -1 and RESULT[1] = mul(e, e in l, type(e,integer))  

  and RESULT[2] = mul(e, e in l, type(e,float))  

  and forall(i::integer, 1<=i and i<=nops(l) and type(l[i],integer) implies l[i]<>0)  

  and forall(i::integer, 1<=i and i<=nops(l) and type(l[i],float) implies l[i]>=0.5))
```

```

or
(1<=status and status<=nops(l)
and RESULT[1] = mul(l[i], i=1..status-1, type(l[i],integer))
and RESULT[2] = mul(l[i], i=1..status-1, type(l[i],float))
and ((type(l[status],integer) and l[status]=0)
or (type(l[status],float) and l[status]<0.5))
and forall(i::integer, 1<=i and i<status and type(l[i],integer) implies l[i]<>0)
and forall(i::integer, 1<=i and i<status and type(l[i],float) implies l[i]>=0.5));
@*)
proc(l::list(Or(integer,float))):[integer,float]; ... end proc;

```

Listing 3. A *MiniMaple* procedure formally specified

The listing gives a formal specification of the example procedure introduced in Section 3. The procedure has no pre-condition as shown in the **requires** clause; the **global** clause says that a global variable *status* can be modified by the body of the procedure. The normal behavior of the procedure is specified in the **ensures** clause.

The post condition specifies that, if the complete list is processed then we get the result as the product of all integers and floats in the list; if the procedure terminates pre-maturely, then we only get the product of integers and floats till the value of variable *status* (index of the input list).

From the example one can also notice the application of numerical quantifier **mul**. The quantifier multiplies only those elements of the input array *l* that satisfy the test **type(e,integer)**.

Loops can be specified by invariants and termination terms denoting non-negative integers as follows:

loop-spec := **invariant** *spec-expr*; **decreases** *spec-expr*;

The following example specifies the loop that iterates over integers from 1...100 respectively computes the sum.

```

i := 1; s := 0; n := 100;
while (i <= n) do{
(*@invariant s = OLD s + i - 1; decreases n-i;@*)
s := s + i; i := i + 1;
}

```

From the example one can see the relationship between the loop variables that holds after every iteration and that the value of the termination term decreases after every iteration.

Loop specifications help in reasoning about loops, i.e. about partial correctness (invariants) and total correctness (termination term).

Assertions have Maple borrowed syntax as given:

asrt := **ASSERT**(*spec-expr*, (EMPTY | “I”));

An assertion can be a logical formula or a named assertion. The following example shows a named assertion (“test failed”).

```
x := 1; y := x; x := x + y;
ASSERT(type(y,integer), "test failed");
```

The implemented type checker also checks the correct typing of the formal specifications. We have used the specification language to formally specify parts of *DifferenceDifferential*. While the specifications are not yet formally checked, they demonstrate the adequacy of the language for the intended purpose.

6 Formal Semantics of *MiniMaple*

We have defined a formal denotational semantics of *MiniMaple* programs as a pre-requisite of a verification calculus which we are currently developing; the verification conditions generated by the verification calculus must be sound with respect to the semantics. There is no formally defined semantics for Maple and only the implementation of Maple can be considered as a basis for our work. However, our semantics definition attempts to depict the internal behavior of Maple. Based on this semantics, now we can ask the question about the correct behavior of any *MiniMaple* program. The complete definition of a formal semantics of *MiniMaple* is given in [15]. Its core features are as follows:

- *MiniMaple* has expressions with side-effects, which is not supported in functional programming languages like Haskell [12] and Miranda [21]. As a result the evaluation of an expression may change the state. The formal semantics of expression evaluation and command execution is therefore defined as a state relationship between pre- and post-states. A formal denotational semantics is defined as a state relationship is easier to integrate with non-uniquely specified procedures as compared to the function-based semantics definition [19].
- Semantic domains of values have some non-standard types of objects, for example symbol, uneval and union etc. *MiniMaple* also supports additional functions and predicates, for example type tests i.e. **type**(E, T), which are correspondingly modeled in semantics algebras.
- In *MiniMaple* a procedure is introduced by an assignment command, e.g. $I := \mathbf{proc}() \dots \mathbf{end\ proc}$, such that assignments take the role of declarations in classical languages. Furthermore, static scoping is used in the definition of a *MiniMaple* procedure.

The denotational semantics is based on *semantic algebra* [18]. For example *Value* is a disjunctive union domain composed of all kinds of primitive semantic values (domains) supported in *MiniMaple*. It also defines some interesting domains i.e. *Module*, *Procedure*, *Uneval* and *Symbol*. The domain *Value* is a recursive domain, e.g. *List* is defined by $Value^*$ as follows:

$$List = Value^*$$

$$Value = \dots | List | \dots$$

A valuation function defines a mapping of a language's abstract syntax structures to its corresponding meaning (semantic algebras) [18]. A valuation function D for a syntax domain D is usually formalized by a set of equations, one per alternative in the corresponding BNF rule for the *MiniMaple* syntactic domain. As the formal semantics of *MiniMaple* is defined as a state relationship so we define the result of valuation function as a predicate. For example the state relation (*StateRelation*) is defined as a power set of pair of pre- and post-states as follows:

$$StateRelation := \mathbb{P}(State \times StateU)$$

The valuation function for command sequence takes the abstract syntax of command sequence, a value of type *Cseq* and type environment *Environment* and results in a *StateRelation* as follows:

$$[[Cseq]] : Environment \rightarrow StateRelation$$

The denotational semantics of *MiniMaple* while-loop is defined as a relationship between a pre-state s and post-state s' as follows:

$$\begin{aligned}
& [[\text{while } E \text{ do } Cseq \text{ end do }]](e)(s,s') \Leftrightarrow \\
& \quad \exists k \in Nat', t, u \in StateU* : t(1) = inStataU(s) \wedge u(1) = inStateU(s) \wedge \\
& \quad (\forall i \in Nat'_k : iterate(i, t, u, e, [[E]], [[Cseq]])) \wedge \\
& \quad \quad ((u(k) = inError() \wedge s' = u(k)) \vee \\
& \quad \quad (returns(data(inState(u(k)))) \wedge s' = t(k)) \vee \\
& \quad \quad (\exists v \in ValueU : [[E]](e)(inState(t(k)), u(k), v) \\
& \quad \quad \quad \wedge v \langle \rangle inValue(inBoolean(True)) \wedge \\
& \quad \quad \quad \text{IF } v = inValue(inBoolean(False)) \text{ THEN} \\
& \quad \quad \quad \quad s' = t(k) \\
& \quad \quad \quad \text{ELSE } s' = inError() \text{ END} \\
& \quad \quad \quad)) \\
& \quad)
\end{aligned}$$

The corresponding *iterate* predicate formalizes the aforementioned while-loop semantics. For the complete list of semantic algebras, domains and valuation functions, please see [15].

7 Conclusions and Future Work

In this paper we gave an overview of *MiniMaple* and its formal type system. We plan to automatically infer types as a future goal. Also we presented our initial work on a formal specification language for *MiniMaple* that can be used to specify the behavior of *MiniMaple* programs. As a main test case we have used our specification language to formally specify various abstract computer algebraic concepts used in the Maple package *DifferenceDifferential*, e.g. difference-differential operator and terms and various related access functions. We may use

this specification language to generate executable assertions that are embedded in *MiniMaple* programs and check at runtime the validity of pre/post conditions. Our main goal, however, is to use the specification language for static analysis, in particular to detect violations of methods preconditions. For this purpose, based on the results of a prior investigation we intend to use the verification framework Why3 [3] to implement the verification calculus for *MiniMaple* as depicted in Fig. 3.

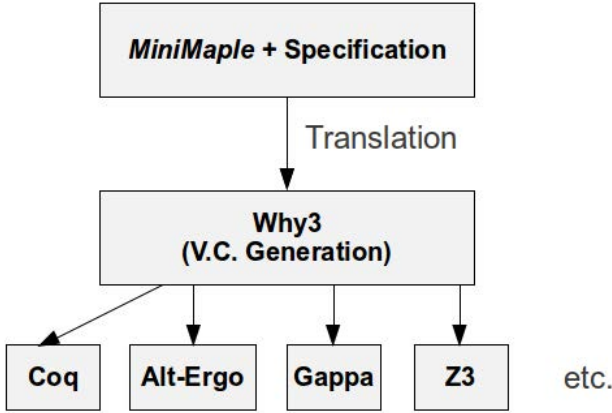


Fig. 3. Verification Calculus for *MiniMaple*

As one can see in the figure, here we need to translate our specification-annotated *MiniMaple* program into the intermediate language of Why3 and then use the various proving back-ends of Why3. Currently we are working on this translation.

References

1. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
2. Baudin, P., Filliâtre, J.C., Hubert, T., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI C Specification Language (preliminary design V1.2), preliminary edn. (May 2008)
3. Bobot, F., Filliâtre, J.-C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, Wrocław, Poland (August 2011)
4. Cardelli, L.: Type Systems. In: Tucker, A.B. (ed.) The Computer Science and Engineering Handbook, pp. 2208–2236. CRC Press (1997)
5. Carette, J., Forrest, S.: Mining Maple Code for Contracts. In: Ranise, S., Bigatti, A. (eds.) Calculemus. Electronic Notes in Theoretical Computer Science. Elsevier (2006)

6. Carette, J., Kucera, M.: Partial Evaluation of Maple. In: Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM 2007, pp. 41–50. ACM Press (2007)
7. Dönch, C.: Bivariate Difference-Differential Dimension Polynomials and Their Computation in Maple. Technical report, Research Institute for Symbolic Computation, Johannes Kepler University, Linz (2009)
8. D'Silva, V., Kroening, D., Weissenbacher, G.: A Survey of Automated Techniques for Formal Software Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27(7), 1165–1178 (2008)
9. Dunstan, M., Kelsey, T., Linton, S., Martin, U.: Lightweight Formal Methods For Computer Algebra Systems. In: International Symposium on Symbolic and Algebraic Computation, ISSAC 1998, pp. 80–87. ACM Press (1998)
10. Leavens, G.T., Cheon, Y.: Design by Contract with JML. A Tutorial (2006), <ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf>
11. Guttag, J.V., Horning, J.J., Garl, W.J., Jones, K.D., Modet, A., Wing, J.M.: Larch: Languages and Tools for Formal Specification. Texts and Monographs in Computer Science. Springer (1993)
12. Hudak, P.: The Haskell School of Expression: Learning Functional Programming through Multimedia. Cambridge University Press (June 2000)
13. Foster, J.S., Furr, M., An, J.-H., Hicks, M.: Static Type Inference for Ruby. In: Proceedings of the 24th Annual ACM Symposium on Applied Computing, OOPS Track, Honolulu, HI (2009)
14. Khan, M.T.: A Type Checker for MiniMaple. RISC Technical Report 11-05, also DK Technical Report 2011-05, Research Institute for Symbolic Computation, Johannes Kepler University, Linz (2011)
15. Khan, M.T.: Formal Semantics of MiniMaple. DK Technical Report 2012-01, Research Institute for Symbolic Computation, Johannes Kepler University, Linz (January 2012)
16. Khan, M.T., Schreiner, W.: On Formal Specification of Maple Programs. In: Conferences on Intelligent Computer Mathematics, Systems and Projects Track (submitted, 2012)
17. Monagan, M.B.: Gauss: A Parameterized Domain of Computation System with Support for Signature Functions. In: Miola, A. (ed.) DISCO 1993. LNCS, vol. 722, pp. 81–94. Springer, Heidelberg (1993)
18. Schmidt, D.A.: Denotational Semantics: a methodology for language development. William C. Brown Publishers, Dubuque (1986)
19. Schreiner, W.: A Program Calculus. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria (September 2008)
20. Boulmé, S., Hardin, T., Hirschhoff, D., Ménissier-Morain, V., Rioboo, R.: On the Way to Certify Computer Algebra Systems. In: Proceedings of the Calculemus Workshop of FLOC 1999 (Federated Logic Conference, Trento, Italie). ENTCS, vol. 23, pp. 370–385. Elsevier (1999)
21. Lambert, T., Lindsay, P., Robinson, K.: Using Miranda as a First Programming Language. *Journal of Functional Programming* 3(1), 5–34 (1993)
22. Zhou, M., Winkler, F.: Computing Difference-Differential Dimension Polynomials by Relative Gröbner Bases in Difference-Differential Modules. *Journal of Symbolic Computation* 43(10), 726–745 (2008)

Formalizing Frankl’s Conjecture: FC-Families

Filip Marić, Miodrag Živković, and Bojan Vučković



Faculty of Mathematics, University of Belgrade*

Abstract. The Frankl’s conjecture, formulated in 1979. and still open, states that in every family of sets closed for unions there is an element contained in at least half of the sets. FC-families are families for which it is proved that every union-closed family containing them satisfies the Frankl’s condition (e.g., in every union-closed family that contains a one-element set a , the element a is contained in at least half of the sets, so families of the form a are the simplest FC-families). FC-families play an important role in attacking the Frankl’s conjecture, since they enable significant search space pruning. We present a formalization of the computer assisted approach for proving that a family is an FC-family. Proof-by-computation paradigm is used and the proof assistant Isabelle/HOL is used both to check mathematical content, and to perform (verified) combinatorial searches on which the proofs rely. FC-families known in the literature are confirmed, and a new FC-family is discovered.

1 Introduction

Formalized mathematics and interactive theorem provers (sometimes referred to as proof assistants) have made great progress in recent years. Many classical mathematical theorems have been formally proved and proof assistants have been intensively used in hardware and software verification. The most successful proof assistants now days are Coq, Isabelle/HOL, HOL Light, etc.

Several of the most important results in formal theorem proving are for the problems that require proofs with much computational content. These proofs are usually highly complex (and therefore often require justifications by formal means) since they combine classical mathematical statements with complex computing machinery (usually computer implementation of combinatorial algorithms). The corresponding paradigm is sometimes referred to as *proof-by-evaluation* or *proof-by-computation*. Probably, the most famous examples of this approach are the proofs of the Four-Color Theorem and the Kepler’s conjecture.

Georges Gonthier has formalized a proof of the Four-Color Theorem  in Coq . The Four Colour Theorem is famous for being the first long-standing mathematical problem, analyzed by many famous mathematicians, finally resolved by

* The first author was partially supported by the Serbian Ministry of Education and Science grant 174021 and by the SNF grant SCOPES IZ73Z0127979/1, the second author by the Serbian Ministry of Education and Science grant 174021 and the third author by the Serbian Ministry of Education and Science grant 044006 (III).

¹ In 1852. Francis Guthrie conjectured that every map can be colored with at most 4 colors such that no two adjacent regions share the same color.

a computer program (Appel and Haken [2]). This proof broke new ground because it involved using IBM 370 assembly language computer programs to carry out a gigantic case analysis, which could not be performed by hand. The proof attracted criticism: computer programming is known to be error-prone, and difficult to relate precisely to the formal statement of a mathematical theorem. Several attempts to simplify the proofs were made (e.g., Robertson et al. [13]), number of cases was reduced and programs were written in C instead of assembly language. However, all doubts were removed only when Gonthier employed proof assistant Coq reducing the whole proof to several basic logical principles.

Another example of a similar kind is the proof of Kepler's conjecture². As described by Nipkow et al. [9]: "In 1998, Thomas Hales announced the first (by now) accepted proof of Kepler's conjecture. It involves 3 distinct large computations. After 4 years of refereeing by a team of 12 referees, the referees declared that they were 99% certain of the correctness of the proof. Dissatisfied with this, Hales started the informal open-to-all collaborative *flyspeck* project to formalize the whole proof with a theorem proof."

In this work, we apply the proof-by-evaluation paradigm to a problem of verifying FC-families — a special case of the Frankl's conjecture. Frankl's conjecture, an elementary and fundamental statement formulated by Péter Frankl in 1979., states that for every family of sets closed under unions, there is an element contained in at least half of the sets (or, dually, in every family of sets closed under intersections, there is an element contained in at most half of the sets). Up to the best of our knowledge, the problem is still open. The conjecture has been proved for many special cases. In particular, it is known to be true for: (i) families of at most 36 sets³ [4]; (ii) families of sets such that their union has at most 11 elements [3].

FC-families are families for which it is proved that all union closed families containing them satisfy the Frankl's condition (if the Frankl's conjecture would be proved, then every family would be an FC-family). For example, it can easily be shown that if a family contains a one-element set, then it satisfies the Frankl's condition. Similar results holds for any two-element set, etc. FC-families are important building block for attempting to prove the Frankl's conjecture since they justify pruning large portions of the search space.

Related work. The Frankl's conjecture has also been formulated and studied as a question in lattice theory [12,11].

FC-families have been introduced by Poonen [11] and further studied by Gao and Yu [5], Vaughan [14,15,16], Morris [8], Marković [7], Bošnjak and Marković [3], and Živković and Vučković [17].

The basic technique used (the Frankl's condition characterization based on weight functions and shares) is introduced by Poonen [11] and later successfully used by Bošnjak and Marković [7,3], and Živković and Vučković [17].

² In 1611 Kepler asserted that the so called cannonball packing is a densest arrangement of 3-dimensional balls of the same size.

³ Unpublished report by Roberts from 1992 claimis a similar result for families of at most 40 sets.

First attempts in using computer-assisted computational approach on solving special cases of the Frankl's conjecture are described by Živković and Vučković [17]. Computations are performed by (unverified) Java programs. However, in order to increase the level of trust, Java programs generate certificates that can be checked by independent tools.

The present paper represent a formalized reformulation of the results of Živković and Vučković [17]. All mathematical content is rigorously formalized within Isabelle/HOL and proofs are mechanically checked. JAVA programs are reimplemented in a functional language of Isabelle/HOL and their correctness is formally verified. A clear separation of mathematical and computational content is done and parts of the proofs that rely on computations are clearly isolated. Since the whole formalization is performed and verified within a proof assistant, there is no need for explicit certificates for statements proved by computation.

Our main contribution are rigorous, machine-verifiable proofs⁴ that all FC-families previously described in the literature are indeed FC-families. Unlike most pen-and-paper proofs, our proofs follow a uniform approach, supported by an underlying combinatorial search procedure. The second contribution is a new type of FC-families: four three-element sets all contained in a seven-element set.

Background logic and notation. Logic and the notation given in this paper will follow Isabelle/HOL. Isabelle/HOL [10] is a development of Higher Order Logic (HOL), and it conforms largely to everyday mathematical notation. The basic types include truth values (*bool*), natural numbers (*nat*) and integers (*int*). Functions can be defined by recursion (either primitive or general). Sets over type α , type α set, follow the usual mathematical conventions⁵. Sets of sets (i.e., object of the type α set set) are called families. Set of all subset for a set A is denoted by `pow A`, and its number of elements is denoted by $|A|$. Lists over type α , type α list, come with the empty list `[]`, the infix prepend constructor `#`, the infix `@` that appends two lists, and the conversion function `set` from lists to sets. N -th element of a list l is denoted by $l_{[n]}$. List $[0, 1, \dots, n - 1]$ is denoted by $[0.. < n]$. The function `sort` sorts a list, `listsum` calculates its sum, and `remdups` removes duplicate elements. List with no repeated elements are called distinct. Standard higher order functions `map`, `filter`, `foldl` are also supported (for details see [10]).

All definitions and statements given in this paper are formalized within Isabelle/HOL. However, in order to make the text accessible to a more general audience not familiar with Isabelle/HOL, many minor details are omitted and some imprecisions are introduced (for example, we used standard symbolics used in related work, although it is clear that some symbols are ambiguous). Statements are grouped into propositions, lemmas, and theorems. Propositions usually express simple, technical results and are printed here without proofs. All sets and families are considered to be finite and this assumptions (present in Isabelle/HOL formalization) will not be explicitly stated in the rest of the paper.

⁴ Corresponding Isabelle/HOL proof documents are available from <http://argo.matf.bg.ac.rs>

⁵ In a strict type setting, sets containing elements of mixed types are not allowed.

Outline. The rest of the paper is organized as follows. In Section 2 we give mathematical background on union-closed families, the Frankl's conjecture and prove main theoretical results. In Section 3 we formulate the combinatorial search algorithm, prove its correctness and give its efficient implementation. In Section 4 we introduce uniform families and techniques used for avoiding symmetries when analyzing them. In Section 5 we verify several kinds of uniform FC-families. Finally, in Section 6 we draw conclusions and give directions for further work.

2 Frankl's Families

2.1 Union Closed Families

First we give basic definitions of union-closed families, closure under unions, and operations used to incrementally obtain closed families.

Definition 1. Let F and F_c be families.

F is union closed, denoted by $uc\ F$, iff $\forall A \in F. \forall B \in F. A \cup B \in F$. F is union closed for F_c , denoted by $uc_{F_c}\ F$, iff $uc\ F \wedge (\forall A \in F. \forall B \in F_c. A \cup B \in F)$.

Closure of F , denoted by $\langle F \rangle$, is the minimal family of sets (in sense of inclusion) that contains F and is union closed. Closure of F for F_c , denoted by $\langle F \rangle_{F_c}$, is the minimal family of sets (in sense of inclusion) that contains F and is union closed for F_c .

Insert and close operation of set A to family F , denoted by $ic\ A\ F$, is the family $F \cup \{A\} \cup \{A \cup B. B \in F\}$. Insert and close operation for F_c of set A to family F , denoted by $ic_{F_c}\ A\ F$, is the family $F \cup \{A\} \cup \{A \cup B. B \in F\} \cup \{A \cup B. B \in F_c\}$.

Proposition 1

1. $\langle F \rangle = \{\bigcup F'. F' \in \text{pow}\ F - \{\emptyset\}\}$
2. $\langle F \cup \{A\} \rangle = ic\ A\ \langle F \rangle, \quad \langle F \cup \{A\} \rangle_I = ic_I\ A\ \langle F \rangle$
3. If $F \subseteq \text{pow}\ \bigcup A$ and $uc_A\ F$ then $uc_{\langle A \rangle}\ F$.

2.2 The Frankl's Condition

The next definition formalizes the Frankl's condition and the notion of FC-family.

Definition 2. Family of sets F satisfies the Frankl's condition and we say that it is a Frankl's family, denoted by $\text{frankl}\ F$, if it contains an element that occurs in at least half sets in the family, i.e., $\text{frankl}\ F \equiv \exists a. a \in \bigcup F \wedge 2 \cdot \#_a F \geq |F|$, where $\#_a F$ denotes $|\{A \in F. a \in A\}|$

Family of sets F_c is FC-family if it is proved that every union closed family such that $F \supseteq F_c$ is Frankl's.

2.3 Family Isomorphisms

The domain of the family does not play any important role for many properties related to the Frankl's condition — many properties are invariant for domain changes using injective functions (that establish a kind of isomorphisms between

two families). Therefore, in many cases it suffices to consider only families over canonical domains — initial ranges $\{0, 1, \dots, n-1\}$ of natural numbers.

Proposition 2. *Let F be a family of sets and f a function injective on $\bigcup F$. Let F' be the image of F under f (then f is a bijection between $\bigcup F$ and $\bigcup F'$).*

1. *If $a \in \bigcup F$, then $\#_a F = \#_{f(a)} F'$.*
2. *$|F| = |F'|$*
3. *If $A \in F$ and $A' \in F'$ is the image of A under f , then $|A| = |A'|$.*
4. *F is union closed if and only if F' is.*
5. *F is Frankl's if and only if F' is.*
6. *If F' is an FC-family, then so is F .*

2.4 FC Characterization by Weight Functions and Shares

We describe the central technique for proving that a family is FC-family, relying on characterizations of the Frankl's condition using weights and shares.

Definition 3. *A function $w : X \rightarrow \mathbb{N}$ is a weight function on $A \subseteq X$, denoted by $\text{wf}_A w$, iff $\exists a \in A. w(a) > 0$. Weight of a set A wrt. weight function w , denoted by $w(A)$, is the value $\sum_{a \in A} w(a)$. Weight of a family F wrt. weight function w , denoted by $w(F)$, is the value $\sum_{A \in F} w(A)$.*

Lemma 1. $\text{frankl } F \iff \exists w. \text{wf}_{(\bigcup F)} w \wedge 2 \cdot w(F) \geq w(\bigcup F) \cdot |F|$

Proof. Assume $\text{frankl } F$ and let a be the element satisfying the Frankl's condition. Let w be the weight function assigning 1 to a and 0 to all other elements. Since $w(F) = \#_a F$ and $w(\bigcup F) = 1$, the statements holds.

Conversely, suppose that $\neg \text{frankl } F$. Then, for every $a \in \bigcup F$, $2 \cdot \#_a F < |F|$. Hence, $2 \cdot w(F) = \sum_{a \in \bigcup F} w(a) \cdot 2 \cdot \#_a F < |F| \cdot \sum_{a \in \bigcup F} w(a) = |F| \cdot w(\bigcup F)$.

A concept that will enable a slightly more operative formulation of the previous characterization is the concept of *share*⁶.

Definition 4. *Let w be a weight function. Share of a set A wrt. w and a set X , denoted by $\bar{w}_X(A)$, is the value $2 \cdot w(A) - w(X)$. Share of a family F wrt. w and a set X , denoted by $\bar{w}_X(F)$, is the value $\sum_{A \in F} \bar{w}_X(A)$.*

Example 1. Let w be a function such that $w(a_0) = 1$, $w(a_1) = 2$, and $w(a) = 0$ for all other elements. w is clearly a weight function. Then, $w(\{a_0, a_1, a_2\}) = 3$ and $w(\{\{a_0, a_1\}, \{a_1, a_2\}, \{a_1\}\}) = 7$. Also, $\bar{w}_{\{a_0, a_1, a_2\}}(\{a_1, a_2\}) = 2 \cdot w(\{a_1, a_2\}) - w(\{a_0, a_1, a_2\}) = 4 - 3 = 1$, and $\bar{w}_{\{a_0, a_1, a_2\}}(\{\{a_0, a_1\}, \{a_1, a_2\}, \{a_1\}\}) = (2 \cdot 3 - 3) + (2 \cdot 2 - 3) + (2 \cdot 2 - 3) = 5$.

Proposition 3. $\bar{w}_X(F) = 2 \cdot w(F) - w(X) \cdot |F|$

Lemma 2. $\text{frankl } F \iff \exists w. \text{wf}_{(\bigcup F)} w \wedge \bar{w}_{(\bigcup F)}(F) \geq 0$

⁶ Note that in order to accommodate for computer implementation only integer weights are allowed, and to avoid rational numbers share of a set A is defined as $2 \cdot w(A) - w(X)$, instead of $w(A) - w(X)/2$ that is used in the literature.

Proof. Follows directly from Proposition 3 and Lemma 1.

Hypercubes. Sets of a family can be grouped into so called hypercubes.

Definition 5. An S -hypercube with a base K , denoted by hc_K^S , is the family $\{A. K \subseteq A \wedge A \subseteq K \cup S\}$. Alternatively, a hypercube can be characterized by $\text{hc}_K^S = \{K \cup A. A \in \text{pow } S\}$.

Example 2. Let $S \equiv \{s_0, s_1\}$, and $K \equiv \{k_0, k_1\}$. If $K' \subseteq K$, then all S -hypercubes with a base K' are:

$$\begin{aligned} \text{hc}_{\{\}}^S &= \{\{\}, \{s_0\}, \{s_1\}, \{s_0, s_1\}\} \\ \text{hc}_{\{k_0\}}^S &= \{\{k_0\}, \{k_0, s_0\}, \{k_0, s_1\}, \{k_0, s_0, s_1\}\} \\ \text{hc}_{\{k_1\}}^S &= \{\{k_1\}, \{k_1, s_0\}, \{k_1, s_1\}, \{k_1, s_0, s_1\}\} \\ \text{hc}_{\{k_0, k_1\}}^S &= \{\{k_0, k_1\}, \{k_0, k_1, s_0\}, \{k_0, k_1, s_1\}, \{k_0, k_1, s_0, s_1\}\} \end{aligned}$$

Previous example indicates that (disjoint) S -hypercubes can span the whole $\text{pow}(K \cup S)$. Indeed, this is generally the case.

Proposition 4. (i) $\text{pow}(K \cup S) = \bigcup_{K' \subseteq K} \text{hc}_{K'}^S$. (ii) If K_1 and K_2 are different and disjoint with S , then $\text{hc}_{K_1}^S$ and $\text{hc}_{K_2}^S$ are disjoint.

Families of sets can be separated into (disjoint) parts belonging to different hypercubes (formed as $\text{hc}_K^S \cap F$).

Definition 6. A hyper-share of a family F wrt. weight function w , the hypercube hc_K^S and the set X , denoted by $\bar{w}_{KX}^S(F)$, is the value $\sum_{A \in \text{hc}_K^S \cap F} \bar{w}_X(A)$.

Example 3. Let S and K be as in the Example 2, let $X \equiv K \cup S$, let $F \equiv \{\{s_0\}, \{s_1\}, \{k_0, s_0\}, \{k_0, k_1, s_0, s_1\}\}$, and $w(a) = 1$ for all $a \in X$. Then, $\bar{w}_{\{\}}^S(F) = \bar{w}_X(\{s_0\}) + \bar{w}_X(\{s_1\}) = -4$, $\bar{w}_{\{k_0\}}^S(F) = \bar{w}_X(\{k_0, s_0\}) = 0$, $\bar{w}_{\{k_1\}}^S(F) = 0$, and $\bar{w}_{\{k_0, k_1\}}^S(F) = \bar{w}_X(\{k_0, k_1, s_0, s_1\}) = 4$.

Share of a family can be expressed in terms of sum of hyper-shares.

Proposition 5. If $K \cup S = \bigcup F$ and $K \cap S = \emptyset$, then $\bar{w}_{(\bigcup F)}(F) = \sum_{K' \subseteq K} \bar{w}_{K'(\bigcup F)}^S(F)$.

Lemma 3. Let w be a weight function on $\bigcup F$. If $K \cup S = \bigcup F$, $K \cap S = \emptyset$, and $\forall K' \subseteq K. \bar{w}_{K'(\bigcup F)}^S(F) \geq 0$, then $\text{frankl } F$.

Proof. Immediate consequence of Proposition 5 and Lemma 2.

Definition 7. Projection of a family F onto a hypercube hc_K^S , denoted by $\text{hc}_K^S \lfloor F$, is the set $\{A - K. A \in \text{hc}_K^S \cap F\}$.

Example 4. Let K, S and F be as in Example 3. Then $\text{hc}_{\{\}}^S \lfloor F = \{\{s_0\}, \{s_1\}\}$, $\text{hc}_{\{k_0\}}^S \lfloor F = \{\{s_0\}\}$, $\text{hc}_{\{k_1\}}^S \lfloor F = \{\}$, and $\text{hc}_{\{k_0, k_1\}}^S \lfloor F = \{\{s_0, s_1\}\}$.

Proposition 6

1. If $K \cap S = \emptyset$ and $K' \subseteq K$, then $\text{hc}_{K'}^S [F] \subseteq \text{pow } S$
2. If $\text{uc } F$, then $\text{uc } (\text{hc}_K^S [F])$.
3. If $\text{uc } F$, $F_c \subseteq F$, $S = \bigcup F_c$, $K \cap S = \emptyset$, then $\text{uc}_{F_c} (\text{hc}_K^S [F])$.
4. If $\forall x \in K. w(x) = 0$, then $\bar{w}_{KX}^S(F) = \bar{w}_X(\text{hc}_K^S [F])$.

Union closed extensions. The next definition introduces an important notion for checking FC-families.

Definition 8. Union closed extensions of a family F_c are families that are created from elements of F_c and are union closed for F_c . Family of all union closed extensions is denoted by $\text{uce } F_c$, and $\text{uce } F_c \equiv \{F' . F' \subseteq \text{pow } \bigcup F_c \wedge \text{uc}_{F_c} F'\}$.

Lemma 4. Let F be a non-empty union closed family, and let F_c be a subfamily (i.e., $F_c \subseteq F$). Let S denote $\bigcup F_c$, and let K denote $\bigcup F - \bigcup F_c$. Let w be a weight function on $\bigcup F$, that is zero for all elements of K . If shares of all union closed extension of F_c are nonnegative, then F is Frankl's, i.e., if $\forall F' \in \text{uce } F_c. \bar{w}_{(\bigcup F_c)}(F') \geq 0$, then $\text{frankl } F$.

Proof. Since, $K \cup S = \bigcup F$ and $K \cap S = \emptyset$, by Lemma 3, it suffices to show that $\forall K' \subseteq K. \bar{w}_{K'(\bigcup F)}^S(F) \geq 0$. Fix K' and assume that $K' \subseteq K$. Since w is zero on K , by Proposition 6, it holds that $\bar{w}_{K'(\bigcup F)}^S(F) = \bar{w}_{(\bigcup F)}(\text{hc}_{K'}^S [F])$. On the other hand, since $\text{uc } F$, $F_c \subseteq F$, and $K \cap S = \emptyset$, by Proposition 6 it holds that $\text{uc}_{F_c} (\text{hc}_{K'}^S [F])$. Moreover, $\text{hc}_{K'}^S [F] \subseteq \text{pow } S$, so $\text{hc}_{K'}^S [F] \in \text{uce } F_c$. Then, $\bar{w}_{(\bigcup F_c)}(\text{hc}_{K'}^S [F]) \geq 0$ holds from the assumption. However, since w is zero on K , it holds that $w(\bigcup F_c) = w(\bigcup F)$ and $\bar{w}_{(\bigcup F)}(\text{hc}_{K'}^S [F]) = \bar{w}_{(\bigcup F_c)}(\text{hc}_{K'}^S [F]) \geq 0$

Theorem 1. A family F_c is an FC-family if there is a weight function w such that shares (wrt. w and $\bigcup F_c$) of all union closed extension of F_c are nonnegative.

Proof. Consider a union-closed family $F \supseteq F_c$. Let w be the weight function such that $\forall F' \in \text{uce } F_c. \bar{w}_{(\bigcup F_c)}(F') \geq 0$. Let w' be a function equal to w on $\bigcup F_c$ and 0 on other elements. Since $\forall F' \in \text{uce } F_c. \bar{w}'_{(\bigcup F_c)}(F') = \bar{w}_{(\bigcup F_c)}(F')$, Lemma 4 applies to F and F is Frankl's.

3 Combinatorial Search

Theorem 1 inspires a procedure for verifying FC families. It should take a weight function on $\bigcup F_c$ and check that all union closed extensions of F_c have non-negative shares. We will now define a procedure *SomeShareNegative*, denoted by $\text{ssn } F_c w$, such that if $\text{ssn } F_c w = \perp$, then for all $F' \in \text{uce } F_c$ it holds that $\bar{w}_{(\bigcup F_c)}(F') \geq 0$. The heart of this procedure will be a recursive function $\text{ssn}^{F_c, w, X} L F_t$ that preforms a systematic traversal of all union closed extensions of F_c , but with pruning that speeds up the search. If a union closed extension of F_c has a negative share, it must contain one or more sets with a negative share. Therefore, a list L of all different subsets of $\bigcup F_c$ with negative shares is formed

and each candidate family is determined by elements of L that it includes. A recursive procedure creates all candidate families by processing elements of L sequentially, either skipping them (in one recursive branch) or including them into the current candidate family F_t (in the other recursive branch), maintaining the invariant that the current candidate family F_t is always union closed. If the current element of L has been already included in F_t (by earlier closure operations required to maintain the invariant) the search can be pruned. If the sum of (negative) shares of the remaining elements of L is less than the (non-negative) share of the current F_t , then F_t cannot be extended to a family with a negative share (even in the extreme case when all the remaining elements of L are included) so, again, the search can be pruned.

Definition 9. *The function $\text{ssn}^{F_c, w, X} L F_t$ is defined by a primitive recursion (over the structure of the list L):*

$$\begin{aligned} \text{ssn}^{F_c, w, X} [] F_t &\equiv \bar{w}_X(F_t) < 0 \\ \text{ssn}^{F_c, w, X} (h \# t) F_t &\equiv \text{if } \bar{w}_X(F_t) + \sum_{A \in h \# t} \bar{w}_X(A) \geq 0 \text{ then } \perp \\ &\quad \text{else if } \text{ssn}^{F_c, w, X} t F_t \text{ then } \top \\ &\quad \text{else if } h \in F_t \text{ then } \perp \\ &\quad \text{else } \text{ssn}^{F_c, w, X} t (\text{ic}_{F_c} h F_t) \end{aligned}$$

Let L be a distinct list such that its set is $\{A. A \in \text{pow } \bigcup F_c \wedge \bar{w}_X(A) < 0\}$.

$$\text{ssn } F_c w \equiv \text{ssn}^{(F_c), w, (\bigcup F_c)} L \emptyset$$

Next we prove the soundness of the $\text{ssn } F_c w$ function.

Lemma 5. *If (i) $\text{ssn}^{F_c, w, X} L F_t = \perp$, (ii) for all elements A in L it holds that $\bar{w}_X(A) < 0$, (iii) for all $A \in F' - F_t$, if $\bar{w}_X(A) < 0$, then A is in L , (iv) $F' \supseteq F_t$, and (v) $\text{uc}_{F_c} F'$, then $\bar{w}_X(F') \geq 0$.*

Proof. The proof is by induction. First, note that

$$\bar{w}_X(F') = \sum_{A \in F'} \bar{w}_X(A) = \sum_{A \in F_t} \bar{w}_X(A) + \sum_{A \in F' - F_t} \bar{w}_X(A). \tag{1}$$

Consider the base case of $L = []$. Since $\text{ssn}^{F_c, w, X} [] F_t = \perp$, it holds that $\sum_{A \in F_t} \bar{w}_X(A) = \bar{w}_X(F_t) \geq 0$ and first term in (1) is nonnegative. If there were some $A \in F' - F_t$ such that $\bar{w}_X(A) < 0$, then, from the assumptions it would be in L , which is impossible since L is empty. Therefore, the second term in (1) is also nonnegative which completes the proof.

Consider the inductive step, and assume that $L \equiv h \# t$.

First consider the case when $\bar{w}_X(F_t) + \sum_{A \in h \# t} \bar{w}_X(A) \geq 0$. Let P denote the set $\{A. A \in F' - F_t \wedge \bar{w}_X(A) \geq 0\}$, and let N denote the set $\{A. A \in F' - F_t \wedge \bar{w}_X(A) < 0\}$. Since, by assumptions, all elements of N are in $L \equiv h \# t$, and since, by assumptions, all shares of $h \# t - N$ are negative, it holds that

$$\sum_{A \in h \# t} \bar{w}_X(A) = \sum_{A \in N} \bar{w}_X(A) + \sum_{A \in h \# t - N} \bar{w}_X(A) \leq \sum_{A \in N} \bar{w}_X(A). \quad (2)$$

It holds that $\sum_{A \in F' - F_t} \bar{w}_X(A) = \sum_{A \in P} \bar{w}_X(A) + \sum_{A \in N} \bar{w}_X(A)$. Therefore, since all shares of P are nonnegative, from (1) and (2) and the assumption of the current case it holds that

$$\bar{w}_X(F') \geq \sum_{A \in F_t} \bar{w}_X(A) + \sum_{A \in N} \bar{w}_X(A) \geq \bar{w}_X(F_t) + \sum_{A \in h \# t} \bar{w}_X(A) \geq 0.$$

Next, consider the case when $\bar{w}_X(F_t) + \sum_{A \in h \# t} \bar{w}_X(A) < 0$. Since, by assumptions, $\text{ssn}^{F_c, w, X}(h \# t) F_t = \perp$, by the definition of ssn it must hold that $\text{ssn}^{F_c, w, X} t F_t = \perp$.

Consider the case when $h \in F_t$ or $h \notin F'$. Then $h \notin F' - F_t$. The conclusion follows by induction hypothesis for the recursive call $\text{ssn}^{F_c, w, X} t F_t$, since all assumptions are satisfied. Indeed, all elements of $F' - F_t$ with negative shares must be in t , since $h \notin F' - F_t$, and other assumptions are trivially satisfied.

Finally, consider the case when $h \notin F_t$ and $h \in F'$. The conclusion follows by induction hypothesis for the recursive call $\text{ssn}^{F_c, w, X} t (\text{ic}_{F_c} h F_t)$, since all assumptions are satisfied for this call. Indeed, in this case $\text{ssn}^{F_c, w, X}(h \# t) F_t = \text{ssn}^{F_c, w, X} t (\text{ic}_{F_c} h F_t)$ and the left hand side is \perp from the current assumptions. All elements of $F' - \text{ic}_{F_c} h F_t$ with negative shares must be in t . Indeed, this holds since $F_t \subseteq \text{ic}_{F_c} h F_t$, and $h \in \text{ic}_{F_c} h F_t$, and since all elements of $F' - F_t$ with negative shares are in $h \# t$. It holds that $\text{ic}_{F_c} h F_t \subseteq F'$ since $F_t \subseteq F'$, $h \in F'$ and $\text{uc}_{F_c} F'$. Other assumptions trivially hold.

Theorem 2. *If $\text{ssn } F_c w = \perp$ and $F' \in \text{uce } F_c$ then $\bar{w}_{(\cup F_c)}(F') \geq 0$.*

Proof. Fix F' from $\text{uce } F_c$. Then $F' \subseteq \text{pow } \cup F_c$ and $\text{uc}_{F_c} F'$. Let L be a distinct list such that its set is $\{A. A \in \text{pow } \cup F_c \wedge \bar{w}_X(A) < 0\}$. From $\text{ssn } F_c w = \perp$ and the definition of ssn it holds that $\text{ssn}^{(F_c), w, (\cup F_c)} L \emptyset = \perp$. All assumptions of Lemma 5 apply. Indeed, for all A in L , $\bar{w}_{(\cup F_c)}(A) < 0$. For all A in $F' - \emptyset$, if $\bar{w}_{(\cup F_c)}(A) < 0$, then, since $F' \subseteq \text{pow } \cup F_c$, A is in L . $\emptyset \subseteq F'$. Since $\text{uc}_{F_c} F'$, by Proposition 1, it holds that $\text{uc}_{(F_c)} F'$. Therefore, $\bar{w}_{(\cup F_c)}(F') \geq 0$ holds.

Apart from being sound, the procedure can also be shown to be complete. Namely, it could be shown that if $\text{ssn } F_c w = \top$, then there is an $F' \in \text{uce } F_c$ such that $\bar{w}_{(\cup F_c)}(F') < 0$. This comes from the invariant that the current family F_t in the search is always in $\text{uce } F_c$, which is maintained by taking the closure $\text{ic}_{F_c} h F_t$ whenever an element h is added. Since this aspect of the procedure is not relevant for the rest of the proofs, it will not be formally stated nor proved.

3.1 Efficient Implementation

In order to obtain executability and increase efficiency, a series of refinements of $\text{ssn } F w$ is done. Each refined version introduces a new implementation feature that makes it more efficient than the previous one, but still equivalent with it.

First, a function cannot operate on families of sets. Without loss of generality, it suffices only to consider families of sets of natural numbers. Sets of natural numbers are represented by natural number codes. A set A is represented by the code $\tilde{A} = \sum_{k \in A} 2^k$. Families of sets of natural numbers F are represented by (distinct) lists of natural number codes \tilde{F} . This representation will be referred to as *list-of-nats* representation (e.g., $F = \{\{0, 1\}, \{1, 2\}, \{0, 1, 2\}\}$ is represented by the list-of-nats $\tilde{F} = [3, 6, 7]$). Basic set operations have their corresponding list-of-nat counterparts.

- The union of two sets \cup corresponds to bitwise disjunction (denoted by \sqcup). It holds that if $C = A \cup B$, then $\tilde{C} = \tilde{A} \sqcup \tilde{B}$.
- Adding a set A to a family of sets F (i.e., $A \cup F$) corresponds to the operation (also denoted by \sqcup) that prepends \tilde{A} to \tilde{F} , but only if it is not already present, i.e., by: if $\tilde{A} \in \tilde{F}$ then \tilde{F} else $\tilde{A} \# \tilde{F}$. It holds that if $F' = A \cup F$, then $\tilde{F}' = \tilde{A} \sqcup \tilde{F}$.
- Union of two families (i.e., $F' \cup F$), also denoted by \sqcup , is performed by iteratively adding sets from one family to another, i.e., as $\text{foldl} (\lambda \tilde{A} \tilde{F}. \tilde{A} \sqcup \tilde{F}) \tilde{F} \tilde{F}'$. It holds that if $F'' = F \cup F'$, then $\tilde{F}'' = \tilde{F} \sqcup \tilde{F}'$.
- Adding a set A to all members of a family of sets F (i.e., $\{A \cup B. B \in F\}$), denoted by $[\tilde{A} \sqcup \tilde{B}. \tilde{B} \in \tilde{F}]$, is performed by $\text{map} (\lambda \tilde{B}. \tilde{A} \sqcup \tilde{B}) \tilde{F}$. It holds that if $F' = \{A \cup B. B \in F\}$, then $\tilde{F}' = [\tilde{A} \sqcup \tilde{B}. \tilde{B} \in \tilde{F}]$.
- Insert and close for F (i.e., $\text{ic}_{F_c} a F$), denoted by $\tilde{\text{ic}}$, is computed as $([\tilde{A}] @ [\tilde{A} \sqcup \tilde{B}. \tilde{B} \in \tilde{F}] @ [\tilde{A} \sqcup \tilde{B}. \tilde{B} \in \tilde{F}_c]) \sqcup \tilde{F}$. It holds that if $F' = \text{ic}_{F_c} a F$, then $\tilde{F}' = \tilde{\text{ic}}_{\tilde{F}_c} \tilde{a} \tilde{F}$.

Important optimization to the basic $\text{ssn } F_c w$ procedure is to avoid repeated computations of family shares (both for the elements of the list L and the current family F_t). So, instead of accepting a list of families of sets L , and the current family of sets F_t , the function is modified to accept a list of ordered pairs where first component is a list-of-nats representation of corresponding element of L , and the second component is its share (wrt. w and X), and to accept an ordered pair (\tilde{F}_t, s_t) where \tilde{F}_t is the list-of-nats representation of F_t , and s_t is its family share (wrt. w and X). The summation of shares of elements in L is also unnecessarily repeated. It can be avoided if the sum (s_l) is passed through the function.

$$\begin{aligned} & \text{ssn}^{\tilde{F}_c, w, X} ([], 0) (\tilde{F}_t, s_t) \equiv s_t < 0 \\ \text{ssn}^{\tilde{F}_c, w, X} ((\tilde{h}, s_h) \# t, s_l) (\tilde{F}_t, s_t) & \equiv \text{if } s_t + s_l \geq 0 \text{ then } \perp \\ & \text{else if } \text{ssn}^{\tilde{F}_c, w, X} (t, s_l - s_h) (\tilde{F}_t, s_t) \text{ then } \top \\ & \text{else if } \tilde{h} \in \tilde{F}_t \text{ then } \perp \\ & \text{else let } \tilde{F}'_t = \tilde{\text{ic}}_{\tilde{F}_c} \tilde{h} \tilde{F}_t; s'_t = \bar{w}_X(\tilde{F}'_t) \text{ in} \\ & \quad \text{ssn}^{\tilde{F}_c, w, X} (t, s_l - s_h) (\tilde{F}'_t, s'_t) \end{aligned}$$

Another source of inefficiency is the calculation of $\bar{w}_X(\tilde{F}'_t)$. If performed directly based on the definition of family share for \tilde{F}'_t , the sum would contain shares of

all elements from \tilde{F}_t and of all elements that are added to \tilde{F}_t when adding \tilde{h} and closing for \tilde{F} . However, it is already known that the sum of shares for elements of \tilde{F}_t is s_t and the implementation could benefit from this fact. Also, calculating shares of sets that are added to \tilde{F}_t can be made faster. Namely, it happens that set share of a same set is calculated over and over again in different parts of the search space. So, it is much better to precompute shares of all sets from $\text{pow } X$ and store them in a lookup table that will be consulted each time a set share is needed. Note that in this case there is no more need to pass the function w itself, nor the domain X , but only the lookup table, denoted by s_w .

$$\begin{aligned}
 & \text{ssn}^{\tilde{F}_c, s_w} ([], 0) (\tilde{F}_t, s_t) \equiv s_t < 0 \\
 \text{ssn}^{\tilde{F}_c, s_w} ((\tilde{h}, s_h) \# t, s_l) (\tilde{F}_t, s_t) & \equiv \text{if } s_t + s_l \geq 0 \text{ then } \perp \\
 & \quad \text{else if } \text{ssn}^{\tilde{F}_c, s_w} (t, s_l - s_h) (\tilde{F}_t, s_t) \text{ then } \top \\
 & \quad \text{else if } \tilde{h} \in \tilde{F}_t \text{ then } \perp \\
 & \quad \text{else } \text{ssn}^{\tilde{F}_c, s_w} (t, s_l - s_h) (\tilde{c}_{\tilde{F}_c}^{s_w} \tilde{h} (\tilde{F}_t, s_t)) \\
 & \tilde{c}_{\tilde{F}_c}^{s_w} \tilde{h} (\tilde{F}_t, s_t) \equiv \text{let } \text{add} = [\tilde{h}] @ [\tilde{h} \sqcup \tilde{A}. \tilde{A} \in \tilde{F}_t] @ [\tilde{h} \sqcup \tilde{A}. \tilde{A} \in \tilde{F}_c]; \\
 & \quad \text{add} = \text{filter } (\lambda \tilde{A}. \tilde{A} \notin \tilde{F}) (\text{remdups add}) \text{ in} \\
 & \quad (\text{add} @ \tilde{F}, s + \text{listsum } (\text{map } s_w \text{ add}))
 \end{aligned}$$

It is shown that this implementation is (in some sense) equivalent to the starting, abstract one. This proof is technically involved, but conceptually uninteresting so we omit it in the text.

4 Uniform *nkm*-Families

Most FC-families that are considered in this paper are *uniform*, i.e., consist of sets having the same number of elements.

Definition 10. *A family of sets F is a uniform nkm -family if it contains m different sets, each containing k elements and their union has at most n elements. Uniform nkm -family is natural if its union is contained in $\{0, 1, \dots, n - 1\}$.*

Within the Isabelle/HOL implementation, natural nkm -families will be represented by *nkm-lists* — (lexicographically) sorted, distinct lists of length m containing sorted, distinct lists of length k with all elements contained in $\{0, 1, \dots, n - 1\}$. To simplify presentation, we will identify natural nkm -families with their corresponding *nkm-lists*. Assuming that the Isabelle/HOL function `comb l k` generates all sorted k -element sublists of a sorted list l , all *nkm-lists* for given n, k and m can be generated by $\text{fams}^{nkm} \equiv \text{comb } (\text{comb } [0.. < n] k) m$.

Symmetries. Often one uniform nkm -family can be obtained from the other by permuting its elements (e.g., $\{\{a_0, a_1, a_2\}, \{a_1, a_3, a_4\}, \{a_2, a_3, a_4\}\}$ can be obtained from $\{\{a_0, a_1, a_2\}, \{a_0, a_1, a_3\}, \{a_2, a_3, a_4\}\}$ by the permutation $(a_0, a_1, a_2, a_3, a_4) \mapsto (a_3, a_4, a_1, a_2, a_0)$). Applying permutations on sets and families can be implemented in Isabelle/HOL by the functions `perm_set A p` \equiv

sort (map (λx. p_[x]) A) and perm_fam F p ≡ sort (map perm_set F). Permutations establish bijections between natural uniform families:

Proposition 7. *If p is a permutation of [0, 1, . . . , n - 1] and F is a natural uniform family, then perm_fam F p is also natural uniform family and there is a bijection between F and perm_fam F p.*

Since, by Proposition 2, FC-families are preserved under bijections (isomorphisms), to check if all elements of a given list of nkm-families \mathcal{F} are FC-families, many elements need not be considered. Indeed, it suffices to consider only a list (denoted by nef^P \mathcal{F}) of its non-equivalent representatives (under a given list of permutations P). Computation of such representatives can start from the given list \mathcal{F} , choose its arbitrary member for a representative, remove it and all its permuted variants from the lists, and repeat this sieving process until the list becomes empty. Isabelle/HOL implementation of this procedure can be given by:

$$\begin{aligned} \text{nef_aux}^P \mathcal{F} \mathcal{F}_r &\equiv \text{case } \mathcal{F} \text{ of } [] \Rightarrow \mathcal{F}_r \\ &| F \# _ \Rightarrow \text{let } \mathcal{F}_F^P = \text{remdups (map (λ p. perm_fam F p) P)} \text{ in} \\ &\quad \text{nef_aux}^P (\text{filter (λ F. F } \notin \mathcal{F}_F^P) \mathcal{F}) (F \# \mathcal{F}_r) \\ \text{nef}^P \mathcal{F} &\equiv \text{nef_aux}^P \mathcal{F} [] \end{aligned}$$

The following lemma proves the correctness of this implementation.

Lemma 6. *If P is a list of permutations of [0, 1, . . . , n - 1] and if \mathcal{F} is a list of natural nkm-families, then for each element F ∈ \mathcal{F} there is an F' ∈ nef^P \mathcal{F} such there is a bijection between F and F'.*

Proof. First, note that the function nef_aux^P $\mathcal{F} \mathcal{F}_r$ is monotone, i.e., $\mathcal{F}_r \subseteq \text{nef_aux}^P \mathcal{F} \mathcal{F}_r$.

By induction, we show that if the assumptions hold for \mathcal{F} and P, then for each element F ∈ \mathcal{F} there is an element F' ∈ nef_aux^P $\mathcal{F} \mathcal{F}_r$ such there is a bijection between F and F'.

In the base case, when \mathcal{F} is empty, the statement trivially holds.

Assume that $\mathcal{F} \equiv F \# \mathcal{F}'$. Let \mathcal{F}_F^P denote all different families obtained by permuting F by all elements of P (i.e., $\mathcal{F}_F^P \equiv \text{remdups (map (λ p. perm_fam F p) P)$) and let \mathcal{F}^- denote what remains of \mathcal{F} when those are removed (i.e., $\mathcal{F}^- \equiv \text{filter (λ F. F } \notin \mathcal{F}_F^P) \mathcal{F}$). It holds that nef_aux^P $\mathcal{F} \mathcal{F}_r = \text{nef_aux}^P \mathcal{F}^- (F \# \mathcal{F}_r)$.

Let F' be an arbitrary element from \mathcal{F} . Since $\mathcal{F} = F \# \mathcal{F}'$, either F' = F or F' ∈ \mathcal{F}' .

Assume that F' = F. By monotonicity it holds that F ∈ nef_aux^P $\mathcal{F} \mathcal{F}_r$, so F is an element from nef_aux^P $\mathcal{F} \mathcal{F}_r$ such that there is a bijection (identity function) between F' and it.

Assume that F' ∈ \mathcal{F}' .

Consider the case when F' ∈ \mathcal{F}_F^P . Then there is p ∈ P such that F' = perm_fam F p. Since F' ∈ \mathcal{F} is natural and p ∈ P is a permutation of

$[0, 1, \dots, n - 1]$, by Proposition 7, there is a bijection between F and F' . Since, by monotonicity, it holds that $F \in \text{nef_aux}^P \mathcal{F} \mathcal{F}_r$, F is an element in $\text{nef_aux}^P \mathcal{F} \mathcal{F}_r$ such that there is a bijection between F' and it.

Consider the case when $F' \notin \mathcal{F}_F^P$. Then $F' \in \mathcal{F}^-$. By inductive hypothesis for the call $\text{nef_aux}^P \mathcal{F}^- (F \# \mathcal{F}_r)$, there is an element F'' in $F \# \mathcal{F}_r$ such that there is a bijection between F' and it. By monotonicity, $F'' \in F \# \mathcal{F}_r \subseteq \text{nef_aux}^P \mathcal{F}^- (F \# \mathcal{F}_r) = \text{nef_aux}^P \mathcal{F} \mathcal{F}_r$, so the statement holds.

Finally, the following lemma shows that only non-equivalent representatives need to be considered when checking FC-families.

Lemma 7. *Let $\mathcal{F} \subseteq \text{fams}^{nkm}$ and $P \subseteq \text{perm} [0, 1, \dots, n - 1]$. If all families represented by elements of $\text{nef}^P \mathcal{F}$ are FC-families, then all families represented by elements of fams^{nkm} are FC-families.*

Proof. Let $F \in \text{fams}^{nkm}$. By Lemma 6 there is an $F' \in \text{nef}^P \mathcal{F}$ and a bijection between F and F' . So, F' is an FC-family, and by Proposition 2, so is F .

5 FC-Families Verified

Having established all the necessary mathematics, in this Section we prove that certain uniform families are FC-families (mainly by performing verified calculations). First, we calculate non-equivalent representatives for fams^{533} , fams^{634} , and fams^{734} .

Lemma 8. *The first column of Table 1 contains (respectively) all elements of:*
 $\text{nef}^{\text{perm} [0..<5]} \text{fams}^{533}$,
 $\text{nef}^{\text{perm} [0..<6]} (\text{filter} (\lambda F. \neg \text{check}_{533} F) \text{fams}^{634})$,
 $\text{nef}^{\text{perm} [0..<7]} (\text{filter} (\lambda F. \neg \text{check}_{533} F \wedge \neg \text{check}_{634} F) \text{fams}^{734})$,

where $\text{perm } l$ is the function that generates all permutations of a list l , check_{533} is a function that checks if any 3 of the 4 given 3-element sets have their union contained in a 5-element set, and check_{634} is a function that checks if the union of 4 given 3-element sets is contained in a 6-element set.⁷

Proof. By calculations performed by a computer.

Next, we show that all these representatives have non-negative shares.

Lemma 9. *For all F_c and w given in Table 1, it holds that $\text{ssn } \tilde{F}_c w = \perp$.*

Proof. By calculations performed by a computer.

Finally, the main result can be easily proved.

⁷ Formal definition of these functions is not given here and is available in the Isabelle/HOL proof documents, along with correctness arguments.

Table 1. Families and weights

F_c	w
$\{[0, 1]\}$	$0 \mapsto 1, 1 \mapsto 1$
$\{[0, 1, 2], [0, 1, 3], [2, 3, 4]\}$	$0 \mapsto 2, 1 \mapsto 2, 2 \mapsto 2, 3 \mapsto 2, 4 \mapsto 1$
$\{[0, 1, 2], [0, 1, 3], [0, 2, 4]\}$	$0 \mapsto 6, 1 \mapsto 5, 2 \mapsto 5, 3 \mapsto 3, 4 \mapsto 3$
$\{[0, 1, 2], [0, 1, 3], [0, 2, 3]\}$	$0 \mapsto 1, 1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 1$
$\{[0, 1, 2], [0, 1, 3], [0, 1, 4]\}$	$0 \mapsto 3, 1 \mapsto 3, 2 \mapsto 2, 3 \mapsto 2, 4 \mapsto 2$
$\{[0, 1, 2], [0, 3, 4], [1, 3, 5], [2, 4, 5]\}$	$0 \mapsto 1, 1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 1, 4 \mapsto 1, 5 \mapsto 1$
$\{[0, 1, 2], [0, 1, 3], [2, 4, 5], [3, 4, 5]\}$	$0 \mapsto 1, 1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 1, 4 \mapsto 1, 5 \mapsto 1$
$\{[0, 1, 2], [0, 3, 4], [1, 3, 5], [2, 4, 6]\}$	$0 \mapsto 2, 1 \mapsto 2, 2 \mapsto 2, 3 \mapsto 2, 4 \mapsto 2, 5 \mapsto 1, 6 \mapsto 1$
$\{[0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5]\}$	$0 \mapsto 2, 1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 1, 4 \mapsto 1, 5 \mapsto 1, 6 \mapsto 1$
$\{[0, 1, 2], [0, 1, 3], [2, 4, 5], [4, 5, 6]\}$	$0 \mapsto 3, 1 \mapsto 3, 2 \mapsto 4, 3 \mapsto 2, 4 \mapsto 3, 5 \mapsto 3, 6 \mapsto 2$
$\{[0, 1, 2], [0, 1, 3], [2, 4, 5], [3, 4, 6]\}$	$0 \mapsto 3, 1 \mapsto 3, 2 \mapsto 3, 3 \mapsto 3, 4 \mapsto 2, 5 \mapsto 1, 6 \mapsto 1$
$\{[0, 1, 2], [0, 1, 3], [0, 4, 5], [4, 5, 6]\}$	$0 \mapsto 6, 1 \mapsto 4, 2 \mapsto 3, 3 \mapsto 3, 4 \mapsto 4, 5 \mapsto 4, 6 \mapsto 2$
$\{[0, 1, 2], [0, 1, 3], [0, 4, 5], [2, 4, 6]\}$	$0 \mapsto 3, 1 \mapsto 2, 2 \mapsto 3, 3 \mapsto 1, 4 \mapsto 3, 5 \mapsto 2, 6 \mapsto 2$
$\{[0, 1, 2], [0, 1, 3], [0, 4, 5], [1, 4, 6]\}$	$0 \mapsto 2, 1 \mapsto 2, 2 \mapsto 1, 3 \mapsto 1, 4 \mapsto 1, 5 \mapsto 1, 6 \mapsto 1$
$\{[0, 1, 2], [0, 1, 3], [0, 4, 5], [0, 4, 6]\}$	$0 \mapsto 2, 1 \mapsto 1, 2 \mapsto 1, 3 \mapsto 1, 4 \mapsto 1, 5 \mapsto 1, 6 \mapsto 1$

Theorem 3. *The following are FC-families:*

1. *all families containing one 1-element set (i.e., $\{\{a\}\}$);*
2. *all families containing one 2-element set (i.e., $\{\{a, b\}\}$, for $a \neq b$);*
3. *all families containing 3 3-element sets whose union is contained in a 5-element set (i.e., uniform 533-families);*
4. *all families containing 4 3-element sets whose union is contained in a 6-element set (i.e., uniform 634-families);*
5. *all families containing 4 3-element sets whose union is contained in a 7-element set (i.e., uniform 734-families).*

Proof. The case 1 trivially holds (since for each family member A that does not contain a , there is a member $A \cup \{a\}$ that contains a).

Other proofs are based on the techniques described in this paper. By Proposition 2 it suffices to consider only families F such that $\bigcup F \subseteq \{0, 1, \dots, n - 1\}$. All families corresponding to rows in Table 1 are FC-families. Indeed, for each F_c and w given in a table row, by Lemma 9 it holds that $\text{ssn } F_c \ w$. Therefore, by Lemma 2 for all $F' \in \text{uce } F_c$ it holds that $\bar{w}_{(\bigcup F_c)}(F') \geq 0$. Then, F_c is FC-family by Theorem 1.

In the case 2 this completes the proof.

In the case 3 the statement holds by Lemma 7, since, by Lemma 8 four rows given in Table 1 correspond to four non-equivalent families.

To show the case 4, let F_c be any family containing 4 3-element sets whose union is contained in $\{0, 1, \dots, 5\}$ and let F be a union-closed family such that $F \supseteq F_c$. If $\text{check}_{533} F_c$ holds (i.e., if union of any 3 members of F_c is contained in a 5-element set), then F is Frankl's by case 3. If $\neg \text{check}_{533} F_c$ holds, then F_c is in filter $(\lambda F. \neg \text{check}_{533} F)$ fams⁶³⁴. The statement then holds by Lemma 7.

since, by Lemma 8 two rows given in Table 11 correspond to two non-equivalent families of filter $(\lambda F. \neg \text{check}_{533} F)$ fams⁶³⁴.

The case 5 is proved similarly, using the proofs for both the case 3 and the case 4.

6 Conclusions and Further Work

In this paper, we have formalized (within Isabelle/HOL) a computer-assisted approach of Živković and Vučković for verifying FC-families. Well-known FC-families are confirmed and a new uniform FC-family is discovered.

The Isabelle/HOL formalization has around 260KB of data organized into around 6500 lines of Isabelle/Isar proof text. Ratio between the size of the formalization and the size of the corresponding pen and paper proof (DeBruijn index) is estimated at around 5.5. Total time required to do the formalization is very roughly estimated at around 200 man/hours (25 full working days spread over a period of around 8 months).

Total proof checking time of Isabelle/HOL takes around 28 minutes on a notebook PC with 2.1GHz Intel/Pentium CPU and 4GB RAM. The major fraction of this time (around 23 minutes) is spent in the combinatorial search. Checking Lemma 9 consumes most of this time, and its last 8 cases (related to the uniform-734 families) alone take 22.8 minutes. This is quite long compared to the original JAVA programs (that perform the whole combinatorial search in around 1 minute), but still bearable. The big difference is due to the use of machine-integers supporting atomic bitwise-or in JAVA and the use of big-integers that do not support atomic bitwise-or in Isabelle/ML. The search time could be reduced if machine-integers were also used in Isabelle/ML. In a simple approach, the code generator could be instructed to replace mathematical integers in the formalization by machine-integers in the code, but that would make a gap between the formalization and the generated code and would require trusting that no overflows occur. A better approach would require formalizing machine-integers and their properties and using them within the formalization itself.

Compared to the prior pen-and-paper work, the computer assisted approach significantly reduces the complexity of mathematical arguments behind the proof and employs computing-machinery in doing its best — quickly enumerating and checking a large search space. This enables formulation of a general framework for checking various FC-families, without the need of employing human intellectual resources in analyzing specificities of separate families. Compared to the work of Živković and Vučković, apart from achieving the highest level of trust possible, the significant contribution of the formalization is the clear separation of mathematical background and combinatorial search algorithms, not present in earlier work. Also, separation of abstract properties of search algorithms and technical details of their implementation significantly simplifies reasoning about their correctness and brings them much closer to classic mathematical audience, not inclined towards computer science.

This work represents a significant part in formally proving the Frankl's conjecture for families F such that $|\bigcup F| \leq 11$, and $|\bigcup F| \leq 12$ (already informally done by Živković and Vučković [17]) which in the focus of our current and future work. We also plan to investigate other FC-families (not necessarily uniform).

References

1. Abe, T.: Strong Semimodular Lattices and Frankl's Conjecture. *Algebra Universalis* 44, 379–382 (2000)
2. Appel, K.I., Haken, W.: Every Planar Map is Four Colorable. American Mathematical Society (1989)
3. Bošnjak, I., Marković, P.: The 11-element Case of Frankl's Conjecture. *Electronic Journal of Combinatorics* 15(1) (2008)
4. Faro, G.L.: Union-closed Sets Conjecture: Improved Bounds. *J. Combin. Math. Combin. Comput.* 16, 97–102 (1994)
5. Gao, W., Yu, H.: Note on the Union-Closed Sets Conjecture. *Ars Combinatorica* 49 (1998)
6. Gonthier, G.: Formal Proof – the Four-Color Theorem. *Notices of AMS* 55(11) (2008)
7. Marković, P.: An attempt at Frankl's Conjecture. *Publications de l'Institut Mathématique* 81(95), 29–43 (2007)
8. Morris, R.: FC-families and Improved Bounds for Frankl's Conjecture. *European Journal of Combinatorics* 27(2), 269–282 (2006)
9. Nipkow, T., Bauer, G., Schultz, P.: Flyspeck I: Tame Graphs. In: Furbach, U., Shankar, N. (eds.) *IJCAR 2006. LNCS (LNAI)*, vol. 4130, pp. 21–35. Springer, Heidelberg (2006)
10. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. *LNCS*, vol. 2283. Springer, Heidelberg (2002)
11. Poonen, B.: Union-closed Families. *Journal of Combinatorial Theory, Series A* 59(2), 253–268 (1992)
12. Reinhold, J.: Frankl's Conjecture is True for Lower Semimodular Lattices. *Graphs and Combinatorics* 16, 115–116 (2000)
13. Robertson, N., Sanders, D.P., Seymour, P.D., Thomas, R.: The Four Colour Theorem. *Journal of Combinatorial Theory, Series B* (1997)
14. Vaughan, T.P.: Families Implying the Frankl Conjecture. *European Journal of Combinatorics* 23(7), 851–860 (2002)
15. Vaughan, T.P.: A Note on the Union-closed Sets Conjecture. *J. Combin. Math. Combin. Comput.* 45, 95–108 (2003)
16. Vaughan, T.P.: Three-sets in a Union-closed Family. *J. Combin. Math. Combin. Comput.* 49, 95–108 (2004)
17. Živković, M., Vučković, B.: The 12-element Case of Frankl's Conjecture (submitted, 2012)

CDCL-Based Abstract State Transition System for Coherent Logic*

Mladen Nikolić and Predrag Janičić

Faculty of Mathematics, University of Belgrade,
Belgrade, Studentski Trg 16, Serbia

Abstract. We present a new, CDCL-based approach for automated theorem proving in coherent logic — an expressive semi-decidable fragment of first-order logic that provides potential for obtaining human readable and machine verifiable proofs. The approach is described by means of an abstract state transition system, inspired by existing transition systems for SAT and represents its faithful lifting to coherent logic. The presented transition system includes techniques from which CDCL SAT solvers benefited the most (backjumping and lemma learning), but also allows generation of human readable proofs. In contrast to other approaches to theorem proving in coherent logic, reasoning involved need not to be ground. We prove key properties of the system, primarily that the system yields a semidecision procedure for coherent logic. As a consequence, the semidecidability of another fragment of first order logic which is a proper superset of coherent logic is also proven.

Keywords: coherent logic, CDCL SAT solving, abstract state transition systems, machine verifiable proofs, readable proofs.

1 Introduction

Coherent logic (CL) is a fragment of first-order logic that involves formulae of the form: $p_1(\vec{v}) \wedge \dots \wedge p_n(\vec{v}) \Rightarrow \exists \vec{y} Q_1(\vec{v}, \vec{y}) \vee \dots \vee \exists \vec{y} Q_m(\vec{v}, \vec{y})$ which are implicitly universally quantified and where $0 \leq n, 0 \leq m$, \vec{v} denotes a sequence of variables v_1, v_2, \dots, v_k , p_i (for $1 \leq i \leq n$) denote atomic formulae (involving some of the variables from \vec{v}), \vec{y} denotes a sequence of variables y_1, y_2, \dots, y_l , and Q_j denote conjunctions of atomic formulae (involving some of the variables from \vec{v} and \vec{y}). If $n = 0$, then the $p_1(\vec{v}) \wedge \dots \wedge p_n(\vec{v})$ part is assumed to be \top , and if $m = 0$, then the $\exists \vec{y} Q_1(\vec{v}, \vec{y}) \vee \dots \vee \exists \vec{y} Q_m(\vec{v}, \vec{y})$ part is assumed to be \perp . There are no function symbols with arity greater than 0.

CL was initially defined by Skolem and in recent years it gained new attention [2, 8, 4, 18]. It allows certain existential quantification, so it is more expressive than the resolution logic. In contrast to the resolution method, the conjecture being proved is kept unchanged and is directly proved (refutation, Skolemization and

* This work was partially supported by the Serbian Ministry of Science grant 174021 and by SNF grant SCOPES IZ73Z0_127979/1.

transformation to clausal form are not used). Hence, proofs in CL are natural and intuitive. In addition, reasoning is constructive and proof objects (verifiable by a proof assistant) can be easily obtained [2,18]. The proof objects in CL also give readable proofs, which is significant for many applications (e.g., in formalizing mathematics and in education). A number of theories and theorems can be formulated directly and simply in CL.

CL is semi-decidable and there are several semi-decision procedures and corresponding theorem provers implemented for it and for similar logics [9,18,2,15]. However, most (although not all) of them are rather simple forward-chaining procedures that can hardly tackle complex conjectures. For such tasks, CL needs more powerful proving engines. We believe that such engine can be based on the dominating CDCL (conflict-driven clause-learning based) approach for SAT solving [5]. A problem with CDCL-based systems is that, in general, they use clausal form, refutation, and Skolemization, so the obtained proofs are not readable (since they are not given in terms of the original signature). In this paper we present one such approach for CL, given in terms of abstract state transition systems, inspired by a transition system for SAT [10]. Our system is a generalization of the system for SAT and can be used as a base both for SAT solving and CL solving. An important distinguishing feature of our system is non-ground reasoning which promises large benefits in practice. Also, we take advantage of nature of CL and design our system so that readable proofs (for instance, in a natural language form or in the Isabelle/Isar form [19]) can be generated. The presented approach, is motivated by and built on the three strong pillars:

Suitability of CL. Coherent logic has a number of good features and is suitable for many automation tasks. It is very expressive and gives potential for obtaining both readable and machine verifiable proofs.

Practical advances in SAT. Over the last decade, a huge progress has been made in SAT solving: a number of high-level algorithmic and low-level implementation features have been developed, so modern SAT solvers can deal with industrial instances with hundreds of thousands of clauses. Our approach should enable the transfer of these advances to coherent logic.

Theoretical advances in SAT. SAT solvers have been described, precisely and suitably for rigorous mathematical analysis, in terms of abstract state transition systems. Their correctness has been proved, first informally [13,10] and then formally (using a proof assistant) [11,12]. These results helped a separation of different concepts used in SAT solvers (often intermixed in typical optimized implementations) and a deeper understanding of operation of SAT solvers. Ideas from transition systems for SAT were used in designing and describing our transition system and for proving its properties (it gives a decision procedure for SAT and a semidecision procedure for CL).

Overview of the paper. The rest of the paper is organized as follows: in Section 2 we give some relevant background information on CL, SAT, and transition systems for SAT; in Section 3 we present our abstract state transition system for CL and in Section 4 we outline its soundness and completeness proofs. In Section 5

we briefly present the mechanism for generating readable proofs based on the presented transition system. In Section 6 we discuss related work, and in Section 7 we draw final conclusions and discuss further work.

2 Background

Propositional logic, First order logic, SAT. We assume the standard notions of propositional and first-order logic (FOL). Propositional logic can be considered as a first-order logic theory with each propositional variable corresponding to a 0-arity predicate symbol. This convention enables considering both propositional logic and coherent logic within the context of first-order logic. For instance, the SAT problem can be defined in the context of FOL, in the following way: SAT is a problem of deciding if it holds $\Gamma \models \perp$ (i.e., whether Γ is unsatisfiable), where Γ is a set of clauses over a signature \mathcal{L} with no function symbols and only with predicate symbols of arity 0. The SAT problem is decidable and is NP-complete.

Abstract State Transition Systems for SAT. The transition system (by Krstić and Goel [10]) given in Figure 1 and referred to as the SAT system hereafter, is used for checking if a set of propositional clauses is satisfiable. It is a terminating, sound and complete (under a certain restrictions) [12]: for any initial state, the system (subject to certain limitations to forget and restart) terminates and then, if C differs from *no_cflct*, then the current formula F is satisfiable (and so is the initial formula F_0) and the current trail M is its model and, otherwise, the current formula F is unsatisfiable (and so is the initial formula F_0).

Coherent logic. The definition of a coherent logic formula is given in Section 1. Coherent logic does not involve negation and the reasoning involved is intuitionistic. For an atomic formula A , $\neg A$ can be represented in the form $A \Rightarrow \perp$, but this translation is not applicable in a general case (for arbitrary formula). In order to reason about negated atomic formulae, for every predicate symbol p , typically a new predicate symbol \bar{p} is introduced that stands for \bar{p} and the following additional axioms are used [17]: $\forall \vec{x}(p(\vec{x}) \vee \bar{p}(\vec{x}))$, $\forall \vec{x}(p(\vec{x}) \wedge \bar{p}(\vec{x}) \Rightarrow \perp)$.

The validity problem in CL is a problem of deciding if it holds $\Gamma \models \Phi$, where Γ is a set of coherent formulae, Φ is a coherent formula, and \models denotes the semantic consequence relation ($\Gamma \models \Phi$ holds if Φ is true in all non-empty Tarskian models for Γ). The validity problem in CL is undecidable, but semidecidable [3].

Any FOL formula can be translated into a CL formula with preserved validity [17]. However, this translation may rely on steps that involve classical logic.

Typically, along a proof of a CL formula, there are new (fresh) constants, *witnesses*, added to the current signature. The term *constant* is used both for the constant symbols from the initial signature and for the witnesses.

3 Abstract State Transition System for CL

In this section we present an abstract state transition system for CL, a base for a semi-decision procedure for CL. It extends and modifies the transition system for SAT given in Section 1. The two systems share the same spirit and the rules

$$\begin{array}{l}
\text{Decide:} \\
\frac{l \in L \quad l, \bar{l} \notin M}{M := M l^d} \\
\text{UnitPropag:} \\
\frac{l \vee l_1 \vee \dots \vee l_k \in F \quad \bar{l}_1, \dots, \bar{l}_k \in M \quad l, \bar{l} \notin M}{M := M l^i} \\
\text{Conflict:} \\
\frac{C = \text{no_cflct} \quad \bar{l}_1 \vee \dots \vee \bar{l}_k \in F \quad l_1, \dots, l_k \in M}{C := \{l_1, \dots, l_k\}} \\
\text{Explain:} \\
\frac{l \in C \quad l \vee \bar{l}_1 \vee \dots \vee \bar{l}_k \in F \quad l_1, \dots, l_k \prec l}{C := C \cup \{l_1, \dots, l_k\} \setminus \{l\}} \\
\text{Learn:} \\
\frac{C = \{l_1, \dots, l_k\} \quad \bar{l}_1 \vee \dots \vee \bar{l}_k \notin F}{F := F \cup \{\bar{l}_1 \vee \dots \vee \bar{l}_k\}} \\
\text{Backjump:} \\
\frac{C = \{l, l_1, \dots, l_k\} \quad \bar{l} \vee \bar{l}_1 \vee \dots \vee \bar{l}_k \in F \quad \text{level } l > m \geq \text{level } l_i}{C := \text{no_cflct} \quad M := M^m \bar{l}} \\
\text{Forget:} \\
\frac{C = \text{no_cflct} \quad c \in F \quad F \setminus c \models c}{F := F \setminus c} \\
\text{Restart:} \\
\frac{C = \text{no_cflct}}{M := M^{[0]}}
\end{array}$$

Fig. 1. Transition system for SAT solving by Krstić and Goel ($l_i \prec l_j$ denotes that the literal l_i precedes l_j in M , l^d denotes a decision literal, l^i an implied literal, level l denotes the decision level of a literal l in M , and M^m denotes the prefix of M up to the level m)

from the SAT system have their counterparts, but typically in a more involved form. In the rest of the paper, the following form of an implicitly universally quantified formula will be considered:

$$\forall \vec{x} p_1(\vec{v}, \vec{x}) \wedge \dots \wedge \forall \vec{x} p_n(\vec{v}, \vec{x}) \Rightarrow \exists \vec{y} q_1(\vec{v}, \vec{y}) \vee \dots \vee \exists \vec{y} q_m(\vec{v}, \vec{y})$$

where atoms p_i involve some of the variables \vec{v} and \vec{x} and atoms q_i involve some of the variables from \vec{v} and \vec{y} . In the rest of the text, by *coherent formula*, we mean a formula of this form. Note that there are two differences from the original coherent form. The first one is that q_i are atoms and not conjunctions of atoms. This restriction does not decrease the expressiveness, since there is a straightforward transformation from the original to this restricted form, since each conjunction can be attributed a new predicate symbol of an appropriate arity, which is then linked, by additional axioms, to that conjunction. Using these axioms, the introduced predicates can be eliminated from the object-level proofs. The second difference from the coherent form is that universal quantifiers may appear in the lefthand side of the formula. This extension can be easily avoided if necessary, but it has some beneficial consequences that we discuss later.

In the rest of the paper we, standardly, do not differentiate between the formulae equal up to renaming of variables. To simplify the presentation, the set of elements of a list L will also be denoted L and the empty list will also be denoted \emptyset . In addition, the set of quantified atoms in the conjunction \mathcal{P} or disjunction \mathcal{Q} will also be denoted \mathcal{P} or \mathcal{Q} . Hence, for a coherent formula $\mathcal{P} \Rightarrow \mathcal{Q}$ where \mathcal{P} is a conjunction of quantified atomic formulae and \mathcal{Q} is a disjunction of quantified atomic formulae, \mathcal{P} and \mathcal{Q} can be also considered as sets. If $\mathcal{P} = \{p_1, \dots, p_n\}$ and $\mathcal{Q} = \{q_1, \dots, q_n\}$, $\forall \vec{x}\mathcal{P}$ will denote $\{\forall \vec{x}p_1, \dots, \forall \vec{x}p_n\}$ and $\exists \vec{y}\mathcal{Q}$ will mean $\{\exists \vec{y}q_1, \dots, \exists \vec{y}q_n\}$. Substitutions will be denoted $\lambda, \sigma, \sigma', \dots$

Definition 1 (Signature and conjecture). *Let V be a countable set of variables and let $\mathcal{L} = (\Sigma^\infty, \Pi, ar)$ be a signature such that $\Sigma^\infty = \{c^i \mid i \in \mathbf{N} \setminus \{0\}\}$, where for each $i = 1, \dots$ it holds $ar(c^i) = 0$, and Π is a finite set of predicates with defined arities. Let no_cflct be a special symbol not appearing in the signature.*

Let there be given a coherent theory \mathcal{T} , i.e., a finite set of coherent axioms \mathcal{AX} , over a signature $(\Sigma_{\mathcal{T}}, \Pi, ar)$ where $\Sigma_{\mathcal{T}} = \{c^1, \dots, c^k\} \subseteq \Sigma^\infty$ and $k \geq 0$, and a conjecture $\forall \vec{x}\mathcal{H}^0(\vec{v}, \vec{x}) \Rightarrow \mathcal{G}^0(\vec{v})$ (over the same signature), where $\mathcal{H}^0(\vec{v}, \vec{x})$ is $h_1^0(\vec{v}, \vec{x}) \wedge \dots \wedge h_m^0(\vec{v}, \vec{x})$ and $\mathcal{G}^0(\vec{v})$ is $\exists \vec{y} g_1^0(\vec{v}, \vec{y}) \vee \dots \vee \exists \vec{y} g_n^0(\vec{v}, \vec{y})$ and h_i^0 and g_i^0 are atomic formulae. Let $\forall \vec{x}\mathcal{H}$ denote $\forall \vec{x}\mathcal{H}^0(\vec{v}, \vec{x})\lambda$, let \mathcal{G} denote $\mathcal{G}^0(\vec{v})\lambda$, for a ground substitution λ over \vec{v} where all constants appearing in λ belong to $\Sigma^\infty \setminus \Sigma_{\mathcal{T}}$ and are pairwise distinct.

In our system, \mathcal{H} will serve as an initial assumption and an element from \mathcal{G} should be reached.

In SAT solving, the model is built incrementally by asserting literals. When both the positive and the corresponding negative literal are asserted the search branch is closed and the backtrack ensues. We define relevant elementary formulae that will take this role in our system.

Definition 2 (Quantified literal). *Positive quantified literal or a quantified atom is a ground atom or $p(\vec{v})$ or $\exists \vec{y} p(\vec{y})$ where p is a predicate symbol and $\exists \vec{y} p(\vec{y})$ is closed. Formulae $\forall \vec{x} p(\vec{v}, \vec{x})$ and $\exists \vec{y} p(\vec{v}, \vec{y})$ are extended quantified atoms or eq-atoms. A formula $(\forall \vec{x} p(\vec{v}, \vec{x})) \Rightarrow \perp$ is a negative quantified literal. A quantified literal is a positive or negative quantified literal. An extended quantified literal or eq-literal is an extended quantified atom or a negative quantified literal. Instead of $l \Rightarrow \perp$, we may write \bar{l} .*

Example 1. In this and the following examples, constants c^1, c^2, \dots will be referred to as a, b, \dots . We will assume $\Pi = \{p, q, r, s\}$ where $ar(p) = ar(q) = ar(r) = 2$ and $ar(s) = 1$. Formulae $p(a, b)$, $p(a, x)$, and $\exists y p(a, y)$ are quantified atoms. Formulae $\forall x p(a, x)$, $\forall x p(v, x)$, and $\exists y p(x, y)$ are extended quantified atoms (due to the presence of universal quantifiers in first two cases, and due to the presence of the free variable x in the third example). Formulae $\overline{p(a, b)}$, $\overline{p(a, x)}$, $\overline{p(x, y)}$, and $\overline{\forall x p(x, y)}$ are negative quantified literals.

Definition 3 (State). A state is a 6-tuple $(\Sigma, \Gamma, M, \mathcal{C}_1, \mathcal{C}_2, \ell)$, where Σ is a finite list of elements from Σ^∞ , Γ is a finite list of coherent formulae over V and (Σ, Π, ar) , M is a list of (pairwise distinct) eq-literals (called a trail) over V and (Σ, Π, ar) , $\mathcal{C}_1 \Rightarrow \mathcal{C}_2$ is a formula called a conflict implication, and ℓ is the index of last introduced constant. An initial state is a state $S_0 = (\Sigma_0, \mathcal{A}\mathcal{X}, \mathcal{H}, \mathcal{G}, \emptyset, \emptyset, n)$, where Σ_0 is the union of constants from $\mathcal{H} \cup \mathcal{G}$ and $\Sigma_{\mathcal{T}}$ (if Σ_0 is empty a constant from Σ^∞ is added in it) and n is the maximal index of constants from Σ_0 .

Intuitively, the role of the state components is as follows: Σ stores the current set of constants, Γ stores the axioms and learnt lemmas, M stores the current set of inferred or assumed eq-literals. The inferred conclusions are logical consequences of the axioms along with assumed quantified literals. The formula $\mathcal{C}_1 \Rightarrow \mathcal{C}_2$ is used in a process called *conflict analysis*.

Definition 4 (Decision levels). The elements of lists M and Σ are divided into decision levels. The elements of different decision levels are separated by the symbol $|$.

The prefix of a list L that includes exactly the elements of the first m decision levels is denoted L^m .

For an element e , $L \frown^m e$ denotes a list obtained from L by inserting e at the end of the decision level m of L if $e \notin L$, and L if $e \in L$. If the number m is omitted, the last level is assumed.

We write $e \in^m L$ if e belongs to the m -th level of L . If $e_i \in^{m_i} L$ for $i = 1, \dots, k$ where $k > 0$, and if $m = \max_i m_i$, then we write $\{e_1, \dots, e_k\} \subseteq^m L$. If $k = 0$, we write $\emptyset \subseteq^0 L$.

We introduce relation denoting precedence of eq-literals in the trail (\prec).

Definition 5 (Relation \prec). We write $l \prec l'$ in some state if it holds $M = M_1 l M_2 l' M_3$ in that state, where any of M_i ($i=1,2,3$) can be empty. For a set S , we write $S \prec l'$ if it holds $l \prec l'$ for each $l \in S$.

Example 2. Let $M = [p(a, b), q(x, y), r(x, y)]$. Then $p(a, b) \prec q(x, y)$. It also holds $\{p(a, b), q(x, y)\} \prec r(x, y)$.

Next, we define the relations of entailment of eq-literals (\Vdash), and validity of eq-literal with respect to the current trail (\Uparrow).

Definition 6 (Relations \Vdash and \Uparrow). For eq-literals l and l' we write $l \Vdash l'$ if there is a substitution λ such that $l\lambda = l'$ and we write $l \Vdash \forall \vec{x} l'$ if $l \Vdash l'$ and we write $l \Vdash \exists \vec{y} l'$ if $l \Vdash l'[\vec{x} \mapsto \vec{t}]$ for some vector of variables and/or constants \vec{t} . We write $S \Vdash S'$ if there exist $l \in S$ and $l' \in S'$ such that $l \Vdash l'$. If some of these sets is singleton, we write its only element instead of it. We write $l \Uparrow^m$ if there is a function m such that $m(l) \in M$ and $m(l) \Vdash l$.

Example 3. It holds $p(x, y) \Vdash p(a, y)$, $p(x, y) \Vdash \forall x p(x, b)$, and $p(x, b) \Vdash \exists y \exists z p(y, z)$. Consequently, if it holds $p(x, b) \in M$, then it holds $\exists y \exists z p(y, z) \Uparrow^m$ for any function m such that $m(\exists y \exists z p(y, z)) = p(x, b)$.

The following two definitions introduce the conflict between eq-literals (\times), and conflict of a formula with the current trail (\downarrow). Note that the term *conflict* is not used in the strict sense of contradiction, but it plays the same role the proper conflict plays in the SAT solving — it signals that backtracking is needed.

Definition 7 (Relation \times). We write $p(\vec{x}, \vec{t}) \times_\lambda \overline{\forall \vec{x}' p(\vec{x}', \vec{t}')}$ (or $\overline{\forall \vec{x}' p(\vec{x}', \vec{t}')} \times_\lambda p(\vec{x}, \vec{t})$) if $\vec{t}\lambda = \vec{t}'\lambda$. We write $\exists y p(\vec{y}, \vec{t}) \times_\lambda p(\vec{x}', \vec{t}') (or $p(\vec{x}', \vec{t}') \times_\lambda \exists y p(\vec{y}, \vec{t})$) if $\vec{t}\lambda = \vec{t}'\lambda$. We write $l \times l'$ if it holds $l \times_\lambda l'$ for some λ .$

Example 4. It holds $\overline{p(a, b) \times p(x, y)}$, $p(x, y) \times \overline{\forall x p(x, b)}$, $p(x, b) \times \overline{\forall x p(x, b)}$, and $\exists x p(x, b) \times p(x, y)$.

Definition 8 (Relation \downarrow). We write $l \downarrow_\lambda^m$ if there is a function m such that $m(l) \in M$ and $l \times_\lambda m(l)$, and we write $l \downarrow$ if it holds $l \downarrow_\lambda^m$ for some m and λ . For a formula $\mathcal{P} \Rightarrow \mathcal{Q}$, a substitution λ , and a partial function $m : \mathcal{P} \cup \mathcal{Q} \rightarrow M$ we write $\mathcal{P} \Rightarrow \mathcal{Q} \downarrow_\lambda^m$ if for each $l \in \mathcal{P}$ it holds $m(l) \Vdash l\lambda$, and for each $l \in \mathcal{Q}$ it holds either $m(l) \times_\lambda l$ or $l\lambda \Vdash \mathcal{G}$ (in the latter case $m(l)$ is not defined). We write $\mathcal{P} \Rightarrow \mathcal{Q} \downarrow$ if it holds $\mathcal{P} \Rightarrow \mathcal{Q} \downarrow_\lambda^m$ for some m and λ . The function m is called the conflict mapping and the set $m(\mathcal{P} \cup \mathcal{Q})$ is called the conflict set for $\mathcal{P} \Rightarrow \mathcal{Q}$. We denote $\{l' \in \mathcal{P} \cup \mathcal{Q} \mid m(l') = l\}$ by $m^{-1}(l)$.

Example 5. Let $\mathcal{G} = \exists x q(a, x) \vee \exists y q(y, b)$ and $M = [p(x, b), \overline{r(a, b)}, s(a)]$. It holds $(p(x, y) \Rightarrow r(x, y)) \downarrow_\lambda^m$ where $m(p(x, y)) = p(x, b)$ and $m(r(x, y)) = r(a, b)$ and $\lambda = [x \mapsto a, y \mapsto b]$. It also holds $(p(x, y) \Rightarrow r(x, y) \vee q(x, b)) \downarrow_\lambda^m$ for the same m and λ (since $q(x, b)\lambda \Vdash \mathcal{G}$).

Negations of \Vdash , \uparrow , and \downarrow are denoted $\not\Vdash$, $\not\uparrow$, and $\not\downarrow$ respectively.

For a fixed set of predicates, the set of all quantified atoms l over the signature Σ for which it holds $l \not\Vdash \mathcal{G}$ is denoted $\mathcal{A}(\Sigma)$. The restriction $l \not\Vdash \mathcal{G}$ on the set $\mathcal{A}(\Sigma)$ is imposed for technical convenience.

The conflict mapping can map several eq-atoms from a formula to the same quantified literal on the trail. We define the merging of these eq-atoms.

Definition 9 (Relation \Rightarrow_λ). Let $l_1 = \forall \vec{x} p(t_1, \dots, t_n)$ and $l_2 = \forall \vec{x}' p(t'_1, \dots, t'_n)$ such that $\vec{x} \cap \vec{x}' = \emptyset$. Let $J = \{i \mid t_i \notin \vec{x} \wedge t'_i \notin \vec{x}'\}$. We write $\{l_1, l_2\} \Rightarrow_\lambda \forall \vec{x}'' p(w_1, \dots, w_n)$ if:

- there is a most general unifier λ for all pairs (t_i, t'_i) ($i \in J$);
- $u_i = t_i\lambda$ and $u'_i = t'_i\lambda$ are not constants if $i \notin J$;
- $w_i = u_i = u'_i$ for $i \in J$ and $w_i = u_i u'_i$ for $i \notin J$ where $u_i u'_i$ is a new variable and $u_i u'_i \in \vec{x}''$.

For $n > 2$, we write $\{l_1, \dots, l_n\} \Rightarrow_\lambda l$ if $\{l_1, l_2\} \Rightarrow_\mu l'$ and $\{l', l_3, \dots, l_n\} \Rightarrow_\nu l$ and $\lambda = \mu\nu$. If λ is not relevant it can be omitted. For eq-literals $\exists \vec{y} p(t_1, \dots, t_n)$ and $\exists \vec{y}' p(t'_1, \dots, t'_n)$, relation \Rightarrow_λ is defined by analogy.

Example 6. Denote $\mathcal{S} = \{\forall x \forall y \phi(x, x, y, y, u, v, w, c), \forall z \phi(z, z, z, z, z, z, t, t)\}$. It holds $\mathcal{S} \Rightarrow_{[w \mapsto c, t \mapsto c]} \forall x z \forall y z \forall u z \forall v z \phi(xz, xz, yz, yz, uz, vz, c, c)$ where c is a constant. It also holds $\{\psi(u, u, v), \psi(v, w, u)\} \Rightarrow_{[u \mapsto v, w \mapsto v]} \psi(v, v, v)$. It does not hold $\{\forall x \psi(x, u, u), \forall y \forall z \psi(y, z, c)\} \Rightarrow_\lambda l$ for any l and any λ .

Definition 10 (CL transition system). For a given (fixed) signature $\mathcal{L} = (\Sigma^\infty, \Pi, ar)$, a CL transition system is a system of rules¹ given in Figure 2. Each rule, when applicable, maps one state to another.

$S \xrightarrow{r} S'$ denotes that state S' can be obtained from state S by the rule r . $S \rightarrow S'$ denotes that $S \xrightarrow{r} S'$ holds for some rule r . A sequence of states S_i such that $S_i \rightarrow S_{i+1}$ is called a chain. A chain is maximal if it is finite and no rule is applicable in its last state or if it is infinite.

Definition 11 (Final states). An accepting state is a state S in which it holds $\mathcal{C}_2 \neq \{no_cflct\}$ and $\mathcal{C}_1 \Rightarrow \mathcal{C}_2 \downarrow^m$ where $m(\mathcal{C}_1 \cup \mathcal{C}_2) \subseteq \mathcal{H}$. In that case, we write $\mathcal{A}\mathcal{X} \vdash_{CL} \forall \vec{x} \mathcal{H}^0(\vec{v}, \vec{x}) \Rightarrow \mathcal{G}^0(\vec{v})$. A rejecting state is a state S for which there is no state S' such that $S \rightarrow S'$ and S is not an accepting state. A state is a final state if it is an accepting state or a rejecting state.

Note that the condition $m(\mathcal{C}_1 \cup \mathcal{C}_2) \subseteq^0 M$ would suffice for the purpose of deciding validity. However, the stronger condition is suitable for purposes of object-level proof generation. We give informal explanations of the rules and some examples.

Decide: This rule makes an assumption. An assumption can be a quantified atom that is not redundant nor contradictory with respect to the current trail (for instance, if there is a quantified literal $s(y)$ on the trail, then the rule is not applicable for $s(c), \exists x s(x)$). Within this step, in M and Σ it is denoted that a new level begins (that depends on the decision made).

Intro: This rule eliminates existential quantifiers and introduces new constant symbols as witnesses. For example, if $M = [p(a, b), \exists y q(x, y)]$, and $\Sigma = a, b$, the rule can insert a fresh constant symbol c into Σ and $q(a, c)$ (or $q(b, c)$) into M .

Unit propagate left/right: If the removal of an atom from a formula results in a conflict of that formula and the trail, that eq-literal can be propagated in positive or negative form depending on the side of the implication it belongs to. If $\underline{p(x, y)} \in M$, a formula $s(x) \Rightarrow \exists y p(x, y)$ can propagate $\underline{s(x)}$. If $\underline{p(a, y)} \in M$, $\underline{s(a)}$ can be propagated. Note that sometimes it is not the propagated eq-literal $\bar{l}\lambda$ that is logically implied, but the eq-literal $(l \Rightarrow \mathcal{G})\lambda$. Still, we propagate $\bar{l}\lambda$ for technical convenience. This does not jeopardize the soundness of our system nor the generation of object-level proofs. Also, note that the propagated eq-literal is inserted into M at the lowest level such that all objects it was derived from are below it.²

Branch end: If $\mathcal{P} \Rightarrow \mathcal{Q} \downarrow$ holds, then the current assumptions are either inconsistent or imply \mathcal{G} , so backtracking is needed. The process of *conflict analysis*³ begins, which aims at finding a lemma that can be used to end subsequent branches that can be ended by derivation analogous to the current one.

¹ Explanations of the rules (we recommend the reader to read them in parallel with the definitions of the rules) and a detailed execution example follow.

² This corresponds to exhaustive unit propagation in SAT solving.

³ The term (somewhat misleading in this context) comes from the SAT solving, where the goal is to reach a contradiction (i.e., a conflict). In CL, the goal is to reach a contradiction *or* the conclusions of a given target formula.

Decide:

$$\frac{l \in \mathcal{A}(\Sigma) \quad l \nmid \quad l \not\downarrow}{M := M|l \quad \Sigma := \Sigma|}$$

Intro:

$$\frac{\exists \bar{y} l \in M \quad (\exists \bar{y} l)\lambda \in \mathcal{A}(\Sigma) \quad l\lambda \nmid \text{ for any } \lambda'}{M := M \wedge l[y_1 \mapsto c^{\ell+1}, \dots, y_k \mapsto c^{\ell+k}]\lambda \quad \Sigma := \Sigma \wedge c^{\ell+1}, \dots, c^{\ell+k} \quad \ell := \ell + k}$$

Unit propagate left:

$$\frac{\mathcal{P} \cup \{l\} \Rightarrow \mathcal{Q} \in^{n_1} \Gamma \quad \mathcal{P} \Rightarrow \mathcal{Q} \downarrow_{\lambda}^m \quad m(\mathcal{P} \cup \mathcal{Q}) \subseteq^{n_2} M \quad \bar{l}\lambda \nmid \quad \bar{l}\lambda \not\downarrow}{M := M \wedge^{\max(n_1, n_2)} \bar{l}\lambda}$$

Unit propagate right:

$$\frac{\mathcal{P} \Rightarrow \mathcal{Q} \cup \{l\} \in^{n_1} \Gamma \quad \mathcal{P} \Rightarrow \mathcal{Q} \downarrow_{\lambda}^m \quad m(\mathcal{P} \cup \mathcal{Q})^{n_2} \subseteq M \quad l\lambda \nmid \quad l\lambda \not\downarrow}{M := M \wedge^{\max(n_1, n_2)} l\lambda}$$

Branch end:

$$\frac{\mathcal{C}_2 = \{no_cflct\} \quad \mathcal{P} \Rightarrow \mathcal{Q} \in \Gamma \quad \mathcal{P} \Rightarrow \mathcal{Q} \downarrow}{\mathcal{C}_1 := \mathcal{P} \quad \mathcal{C}_2 := \mathcal{Q}}$$

Explain left \forall :

$$\frac{\mathcal{C}_1 \Rightarrow \mathcal{C}_2 \downarrow^m \quad l \in m(\mathcal{C}_1) \quad S = m^{-1}(l) \quad S \Rightarrow \forall \bar{x} p(\bar{v}, \bar{x})}{\mathcal{P} \Rightarrow \mathcal{Q} \cup \{p(\bar{v}', \bar{x}')\} \in \Gamma \quad \mathcal{P} \Rightarrow \mathcal{Q} \downarrow^{m'} \quad m'(\mathcal{P} \cup \mathcal{Q}) \prec l \quad \overline{\forall \bar{x} p(\bar{v}, \bar{x})} \times_{\lambda} p(\bar{v}', \bar{x}')}{\mathcal{C}_1 := (\forall \bar{x}' \mathcal{P} \cup (\mathcal{C}_1 \setminus S))\lambda \quad \mathcal{C}_2 := (\exists \bar{x}' \mathcal{Q} \cup \mathcal{C}_2)\lambda}$$

Explain left \exists :

$$\frac{\mathcal{C}_1 \Rightarrow \mathcal{C}_2 \downarrow^m \quad l \in m(\mathcal{C}_1) \quad S = m^{-1}(l) \quad S \Rightarrow_{\sigma} p(\bar{v}, \bar{x})}{\mathcal{P} \Rightarrow \mathcal{Q} \cup \{\exists \bar{x}' p(\bar{v}', \bar{x}')\} \in \Gamma \quad \mathcal{P} \Rightarrow \mathcal{Q} \downarrow^{m'} \quad m'(\mathcal{P} \cup \mathcal{Q}) \prec l \quad \overline{p(\bar{v}, \bar{x})} \times_{\lambda} \exists \bar{x}' p(\bar{v}', \bar{x}')}{\mathcal{C}_1 := (\mathcal{P} \cup \forall \bar{x} (\mathcal{C}_1 \sigma \setminus S \sigma))\lambda \quad \mathcal{C}_2 := (\mathcal{Q} \cup \exists \bar{x} (\mathcal{C}_2 \sigma))\lambda}$$

Explain right \forall :

$$\frac{\mathcal{C}_1 \Rightarrow \mathcal{C}_2 \downarrow^m \quad l \in m(\mathcal{C}_2) \quad S = m^{-1}(l) \quad S \Rightarrow_{\sigma} p(\bar{v}, \bar{x})}{\{\forall \bar{x}' p(\bar{v}', \bar{x}')\} \cup \mathcal{P} \Rightarrow \mathcal{Q} \in \Gamma \quad \mathcal{P} \Rightarrow \mathcal{Q} \downarrow^{m'} \quad m'(\mathcal{P} \cup \mathcal{Q}) \prec l \quad \overline{p(\bar{v}, \bar{x})} \times_{\lambda} \overline{\forall \bar{x}' p(\bar{v}', \bar{x}')}}}{\mathcal{C}_1 := (\mathcal{P} \cup \forall \bar{x} (\mathcal{C}_1 \sigma))\lambda \quad \mathcal{C}_2 := (\mathcal{Q} \cup \exists \bar{x} (\mathcal{C}_2 \sigma \setminus S \sigma))\lambda}$$

Explain right \exists :

$$\frac{\mathcal{C}_1 \Rightarrow \mathcal{C}_2 \downarrow^m \quad l \in m(\mathcal{C}_2) \quad S = m^{-1}(l) \quad S \Rightarrow \exists \bar{x} p(\bar{v}, \bar{x})}{\{p(\bar{v}', \bar{x}')\} \cup \mathcal{P} \Rightarrow \mathcal{Q} \in \Gamma \quad \mathcal{P} \Rightarrow \mathcal{Q} \downarrow^{m'} \quad m'(\mathcal{P} \cup \mathcal{Q}) \prec l \quad \exists \bar{x} p(\bar{v}, \bar{x}) \times_{\lambda} \overline{p(\bar{v}', \bar{x}')}}}{\mathcal{C}_1 := (\forall \bar{x}' \mathcal{P} \cup \mathcal{C}_1)\lambda \quad \mathcal{C}_2 := (\exists \bar{x}' \mathcal{Q} \cup (\mathcal{C}_2 \setminus S))\lambda}$$

Learn:

$$\frac{\mathcal{C}_2 \neq \{no_cflct\} \quad \mathcal{C}_1 \Rightarrow \mathcal{C}_2 \notin \Gamma}{\Gamma := \Gamma \wedge \mathcal{C}_1 \Rightarrow \mathcal{C}_2}$$

Backjump:

$$\frac{\mathcal{C}_1 \Rightarrow \mathcal{C}_2 \in \Gamma \quad \mathcal{C}_1 \Rightarrow \mathcal{C}_2 \downarrow^m \quad l \in m(\mathcal{C}_1) \quad S = m^{-1}(l) \quad \mathcal{C}_1 \setminus S \Rightarrow \mathcal{C}_2 \downarrow_{\lambda}^{m'} \quad m' \subseteq m \quad m'(\mathcal{C}_1 \setminus S \cup \mathcal{C}_2) \subseteq^n M \quad l \in^{n'} M \quad n \leq t < n' \quad S\lambda \Rightarrow l'}{M := M^{t \wedge n'} \quad \Sigma := \Sigma^t \quad \mathcal{C}_1 := \emptyset \quad \mathcal{C}_2 := \{no_cflct\}}$$

Fig. 2. Abstract state transition system for CL

Explain left/right \forall/\exists : These rules perform conflict analysis by performing a kind of generalized resolution on the conflict implication and formulae from Γ that (could have) propagated quantified literals in the conflict set. The resolution can be described by following schematic rules, but in the Explain rules it is adjusted for resolving several literals at once when several eq-atoms from conflict implication correspond to the same quantified literal in the conflict set.

$$\frac{\mathcal{P} \Rightarrow \mathcal{Q} \cup \{\exists \vec{y} p(\vec{x}, \vec{y})\} \quad \{p(\vec{x}', \vec{y}')\} \cup \mathcal{P}' \Rightarrow \mathcal{Q}'}{(\mathcal{P} \cup \forall \vec{y}' \mathcal{P}' \Rightarrow \mathcal{Q} \cup \exists \vec{y}' \mathcal{Q}') \lambda} \quad \frac{\mathcal{P} \Rightarrow \mathcal{Q} \cup \{p(\vec{x}, \vec{y})\} \quad \{\forall \vec{x}' p(\vec{x}', \vec{y}')\} \cup \mathcal{P}' \Rightarrow \mathcal{Q}'}{(\forall \vec{x} \mathcal{P} \cup \mathcal{P}' \Rightarrow \exists \vec{x} \mathcal{Q} \cup \mathcal{Q}') \sigma}$$

where λ is the most general unifier for \vec{x} and \vec{x}' and σ is the most general unifier for \vec{y} and \vec{y}' . Notice that if length of \vec{y} in the first case and \vec{x}' in the second case is 0, the two rules are the same and in such case it is not important which one is used. Suppose a conflict implication is $p(x, y) \wedge q(x, y) \Rightarrow r(x, y)$ and that there is an axiom $s(x) \Rightarrow \exists y p(x, y)$. If the Explain left \exists is applied to obtain a new conflict implication by resolving these two, $s(x) \wedge \forall y q(x, y) \Rightarrow \exists y r(x, y)$ is obtained. Now, suppose that the conflict implication is $p(v, z) \wedge \forall x q(x, v) \wedge \forall x q(v, x) \Rightarrow \exists y r(z, y)$, where both q eq-atoms in its lefthand side correspond to the same quantified atom in the conflict set, and that there is an axiom $p(x, y) \Rightarrow q(x, y)$. If Explain left \forall is applied, since it holds $\{\forall x q(x, v), \forall x q(v, x)\} \Rightarrow \forall u \forall w q(u, w)$, the new conflict implication is $(\forall v p(v, z) \wedge \forall x \forall v p(x, v)) \Rightarrow \exists y r(z, y)$.

Learn: In the conflict analysis process, an implication $\mathcal{C}_1 \Rightarrow \mathcal{C}_2$ is derived. Since it is a consequence of the axioms, it can be added to Γ as a learnt lemma.

Backjump: Since the conflict implication $\mathcal{C}_1 \Rightarrow \mathcal{C}_2$ is in conflict with M , some of the quantified literals have to be removed from the trail, so backjumping is performed. The level of the backjump is chosen so that only the quantified literal l from the top level in the conflict set is removed. Also, since other quantified literals from the conflict set are still present on the trail, a negative quantified literal can be derived from $\mathcal{C}_1 \Rightarrow \mathcal{C}_2$ that prevents l from appearing on the trail again. The last condition in the rule is concerned with the case when there are several quantified literals in \mathcal{C}_1 that correspond to l .

Example 7. Let us illustrate the operation of the CL transition system on the following (artificial) example. Let the axioms \mathcal{AX} of \mathcal{T} be (implicitly universally quantified) formulae: (Ax1) $p(x, y) \wedge q(x, y) \wedge r(x, y) \Rightarrow \perp$, (Ax2) $s(x) \Rightarrow \exists y q(x, y)$, (Ax3) $q(x, y) \Rightarrow r(x, y)$, and (Ax4) $s(x) \vee q(y, y)$, and the conjecture is $\forall z p(x, z) \Rightarrow \perp$.

The free variable of the conjecture is instantiated by fresh constant a : $\mathcal{H} = p(a, z)$ and $\mathcal{G} = \perp$. The initial state is $S_0 = (\{a\}, \mathcal{AX}, p(a, z), \emptyset, \emptyset, 1)$. The details of operation are given in the following table (note that the order in which the rules are applied is not fixed). Since the last state is an accepting state, the theorem has been proved.

Rule applied	Σ	$\Gamma \setminus \mathcal{AX}$ (lemmas)	M	$\mathcal{C}_1 \Rightarrow \mathcal{C}_2$
Decide	a	\emptyset	$p(a, z)$	$\emptyset \Rightarrow \{\text{no_cflct}\}$
U. p. right (Ax2)	a	\emptyset	$p(a, z) s(a)$	$\emptyset \Rightarrow \{\text{no_cflct}\}$
Intro	a	b	$p(a, z) s(a), \exists y q(a, y)$	$\emptyset \Rightarrow \{\text{no_cflct}\}$
U. p. left (Ax1)	a	b	$p(a, z) s(a), \exists y q(a, y), q(a, b)$	$\emptyset \Rightarrow \{\text{no_cflct}\}$
Branch end (Ax3)	a	b	$p(a, z) s(a), \exists y q(a, y), q(a, b), \overline{r(a, b)}$	$\emptyset \Rightarrow \{\text{no_cflct}\}$
Ex. right \forall/\exists (Ax1)	a	b	$p(a, z) s(a), \exists y q(a, y), q(a, b), \overline{r(a, b)}$	$q(x, y) \Rightarrow r(x, y)$
Ex. left \exists (Ax2)	a	b	$p(a, z) s(a), \exists y q(a, y), q(a, b), \overline{r(a, b)}$	$p(x, y) \wedge q(x, y) \Rightarrow \perp$
Learn	a	b	$p(a, z) s(a), \exists y q(a, y), q(a, b), \overline{r(a, b)}$	$\forall y p(x, y) \wedge s(x) \Rightarrow \perp$
Backjump	a	\emptyset	$p(a, z), \overline{s(a)}$	$\forall y p(x, y) \wedge s(x) \Rightarrow \perp$
U. p. right (Ax4)	a	\emptyset	$p(a, z), \overline{s(a)}, q(y, y)$	$\emptyset \Rightarrow \{\text{no_cflct}\}$
U. p. left (Ax1)	a	\emptyset	$p(a, z), \overline{s(a)}, q(y, y), \overline{r(a, a)}$	$\emptyset \Rightarrow \{\text{no_cflct}\}$
Branch end (Ax3)	a	\emptyset	$p(a, z), \overline{s(a)}, q(y, y), \overline{r(a, a)}$	$\emptyset \Rightarrow \{\text{no_cflct}\}$
Ex. right \forall/\exists (Ax1)	a	\emptyset	$p(a, z), \overline{s(a)}, q(y, y), \overline{r(a, a)}$	$q(x, y) \Rightarrow r(x, y)$
Ex. left (Ax4)	a	\emptyset	$p(a, z), \overline{s(a)}, q(y, y), \overline{r(a, a)}$	$p(x, y) \wedge q(x, y) \Rightarrow \perp$
Ex. right (lemma)	a	\emptyset	$p(a, z), \overline{s(a)}, q(y, y), \overline{r(a, a)}$	$p(x, x) \Rightarrow s(z)$
				$p(x, x) \wedge \forall u p(z, u) \Rightarrow \perp$

As already noted, the fragment we are working with is syntactically broader than CL due to the presence of universal quantifiers in the lefthand side of the formulae. This extension allows more expressive lemma learning mechanism, compliant with the use of (implicitly) universally quantified literals on the trail. If one wants to stay within the original CL fragment, three conditions should be fulfilled. The rule Decide should insert only closed quantified atoms on the trail. The Explain rules should choose the literal l from the conflict set such that all other literals from the set precede it on the trail and should resolve the conflict implication with an axiom (or a learnt lemma) that was used to propagate (or derive in the backjump) the literal l . Finally, if there are universal quantifiers in the left side of the conflict implication, the Explain rules should be applied until these quantifiers are removed.

4 Properties of CL Transition Systems

In this section we state the properties of the proposed system. Both soundness and completeness are proven with respect to non-empty Tarskian models of the axiom set. The relation \models denotes a logical consequence. Full proofs of the theorems are available in the appendix.⁴

The soundness of the transition system given in Figure 2 is proven by showing that if an accepting state is reached for a coherent formula Φ and a coherent theory \mathcal{AX} , then Φ is a logical consequence of \mathcal{AX} .

Theorem 1 (Soundness). *If it holds $\mathcal{AX} \vdash_{CL} \forall \vec{x} \mathcal{H}^0(\vec{v}, \vec{x}) \Rightarrow \mathcal{G}^0(\vec{v})$, then it holds $\mathcal{AX} \models \forall \vec{x} \mathcal{H}^0(\vec{v}, \vec{x}) \Rightarrow \mathcal{G}^0(\vec{v})$.*

Proof outline. Since the accepting state can be reached, there is a conflict implication $\mathcal{C}_1 \Rightarrow \mathcal{C}_2$ and its corresponding resolution proof (in the sense of generalized resolution defined by the Explain rules). Resolution steps defined by the Explain rules are sound, and they involve only the axioms and previously learnt lemmas (derived again from the axioms), so $\mathcal{C}_1 \Rightarrow \mathcal{C}_2$ is a logical consequence of \mathcal{AX} . By the definition of an accepting state, the conflict set \mathcal{S} for $\mathcal{C}_1 \Rightarrow \mathcal{C}_2$ is a subset of \mathcal{H} . Moreover, by the definition of the conflict set, \mathcal{S} contains positive quantified literals that satisfy all conjuncts from $\mathcal{C}_1\lambda$ for some λ . Also, each disjunct from \mathcal{C}_2 either implies \mathcal{G} or corresponds to a negative quantified literal from \mathcal{S} . Since $\mathcal{S} \subseteq \mathcal{H}$ and \mathcal{H} has no negative quantified literals, in \mathcal{C}_2 there can be only disjuncts that imply \mathcal{G} . Therefore, $\mathcal{C}_1 \Rightarrow \mathcal{G}$ is a logical consequence of \mathcal{AX} , so $(\mathcal{C}_1 \Rightarrow \mathcal{G})\lambda$ and (since \mathcal{G} is closed) $\mathcal{C}_1\lambda \Rightarrow \mathcal{G}$ are logical consequences of \mathcal{AX} . If \mathcal{H} is true in some model of \mathcal{AX} , then $\mathcal{C}_1\lambda$ is true in that model, and consequently \mathcal{G} , too.

There are no guarantees that an arbitrary order of rule applications for a valid formula leads to the accepting state. In order to ensure this property, that we call *strong completeness*, we introduce the following restrictions to our system.

⁴ The appendix is available online from <http://argo.matf.bg.ac.rs/cdclcl.pdf>

We extend the state with a list of numbers Δ and a number δ . So, a state is a 8-tuple $(\Sigma, \Gamma, M, \mathcal{C}_1, \mathcal{C}_2, \ell, \Delta, \delta)$. In an initial state S_0 , it holds $\delta_0 = \ell_0$ and $\Delta_0 = \delta_0$. Also, we add the *limiter rule*:

Limiter:

$$\frac{\text{No other rules applicable} \quad \delta < \max\{i \mid c^i \in \Sigma\}}{\delta := \delta + 1}$$

The indices of constants in the quantified atom l in Decide and Intro rule have to be less than or equal to δ . The same holds for substitution λ in Backjump and Unit propagate. Effects of the Decide rule are extended by $\Delta := \Delta \mid \delta$ and of the Backjump rule by $\Delta := \Delta^m$ and $\delta := \delta'$ where δ' is the last element of Δ^m . Although weaker restriction can be made, we restrict the Explain rules to choose the literal l from the conflict set such that all other literals from the set precede it on the trail, and to resolve the conflict implication with an axiom (or a learnt lemma) that was used to propagate (or derive in the backjump) the literal l . These formulae (called reason clauses in SAT solving) can be easily found if the record is kept when Unit propagation and Backjump are applied, as is the common practice in SAT solving. Note that the new system is a restriction of the original one, so the soundness arguments hold for the new system, too. The strong completeness can be proven — that for a valid formula, any order of rule applications (compliant with the imposed restrictions) will reach an accepting state.

Theorem 2 (Strong completeness). *If it holds $\mathcal{AX} \models \forall \vec{x} \mathcal{H}^0(\vec{v}, \vec{x}) \Rightarrow \mathcal{G}^0(\vec{v})$, then in each maximal chain $S_0 \rightarrow \dots$ there exists an accepting state.*

Proof outline. Suppose that there is no accepting state in the chain. Then, the chain is either infinite or ends in a state in which no rule is applicable. Let L be the set of all quantified atoms that are permanently kept on the trail after some state in the chain. Consider a ground model \mathcal{M} in which a ground atom l is true if $L \Vdash l$. If the premises of an axiom are true in \mathcal{M} , it can be shown that its conclusions are also true in \mathcal{M} . So, \mathcal{M} is a model for \mathcal{AX} . \mathcal{H} is trivially true in \mathcal{M} . Let us suppose that in some state S it holds $g \uparrow$ for some $g \in \mathcal{G}$. This indicates a branch end, and it can be shown that a backjump follows, after which $g \not\uparrow$ holds. Hence \mathcal{G} is not true in \mathcal{M} . This shows that \mathcal{M} is not a model for $(\forall * \mathcal{H}) \Rightarrow \mathcal{G}$ even though it is a model for \mathcal{AX} which is a contradiction with the assumptions of the statement.

5 Generation of Readable Proofs

CDCL-based systems typically provide resolution refutation proofs that are not readable because transformation to clausal form, refutation and Skolemization are used. These proofs can, in principle, be transformed to forward chaining proofs, but these proofs would be hardly readable (because of the transformed axioms and function symbols non-existent in the original theory) and would not resemble textbook proofs (e.g., in geometry). This is avoided in our system,

enabling generation of forward chaining proofs that can serve for simple building of readable proofs (for instance, in a natural language form or in the Isabelle/Isar form). We define a coherent forward chaining proof system (inspired by proof system given in [17]) as follows. The axioms are $\Gamma, \perp \vdash \Phi$ and $\Gamma, \phi \vdash \Phi$ if $\phi \Vdash \Phi$. The rules are:

$$\frac{\Gamma, A, A \Rightarrow B, B \vdash \Phi}{\Gamma, A, A \Rightarrow B \vdash \Phi} \Rightarrow \quad \frac{\Gamma, A \vdash \Phi \quad \Gamma, B \vdash \Phi}{\Gamma, A \vee B \vdash \Phi} \vee \quad \frac{\Gamma, A, A\sigma \vdash \Phi}{\Gamma, A \vdash \Phi} \text{Inst}$$

$$\frac{\Gamma, p(\vec{x}), \forall \vec{x} p(\vec{x}) \vdash \Phi}{\Gamma, p(\vec{x}) \vdash \Phi} \forall \quad \frac{\Gamma, \exists \vec{y} p(\vec{x}, \vec{y}), p(\vec{a}, \vec{c}) \vdash \Phi}{\Gamma, \exists \vec{y} p(\vec{x}, \vec{y}) \vdash \Phi} \exists \quad \frac{\Gamma, p(\vec{a}), p(\vec{x}) \vdash \Phi}{\Gamma, p(\vec{a}) \vdash \Phi} \text{Eigen}$$

where in \vee rule A and B share no free variables, in Inst, σ is an arbitrary substitution over variables and constants from Σ^∞ , in \exists rule \vec{a} consists of constants appearing in Γ and $\exists \vec{y} p(\vec{x}, \vec{y})$ and \vec{c} consists of fresh constants, and Eigen can be applied to constants from \vec{a} only in one branch of the proof and those constants must have been introduced by Inst as fresh constants at the step in which they first appear (this is a kind of eigenvariable condition). For a formula $\forall \vec{x} \mathcal{H}^0(\vec{v}, \vec{x}) \Rightarrow \mathcal{G}^0(\vec{v})$ the coherent forward chaining proof is a derivation tree for $\mathcal{A}\mathcal{X}, \mathcal{H} \vdash \mathcal{G}$.

Theorem 3. *If $\mathcal{A}\mathcal{X} \vdash_{CL} \forall \vec{x} \mathcal{H}^0(\vec{v}, \vec{x}) \Rightarrow \mathcal{G}^0(\vec{v})$, there exists a coherent forward chaining proof for $\forall \vec{x} \mathcal{H}^0(\vec{v}, \vec{x}) \Rightarrow \mathcal{G}^0(\vec{v})$.*

Proof outline. Since the accepting state can be reached, there is a conflict implication $\mathcal{C}_1 \Rightarrow \mathcal{C}_2$ and its resolution proof built from the axioms. A proof of the conjecture can be generated recursively from this resolution proof. Let resolving an eq-literal from the lefthand side of some \mathcal{F}_1 and an eq-literal from the righthand side of some \mathcal{F}_2 yield a conflict implication $\mathcal{C}_1 \Rightarrow \mathcal{C}_2$. If Pr_1 and Pr_2 are forward chaining proofs for \mathcal{F}_1 and \mathcal{F}_2 , then the proof of the conjecture is constructed (roughly) by replacing non-contradiction leaves of Pr_1 by Pr_2 (with appropriate renaming of fresh constants and free variables in Pr_2).

A generated proof need not be axiom-level, but can involve learnt lemmas with their proofs generated separately.

Example 8. We present the resolution tree for last derived $\mathcal{C}_1 \Rightarrow \mathcal{C}_2$, corresponding to the example of system execution given in Example 7.

$$\frac{\frac{\frac{q(x, y) \Rightarrow r(x, y) \quad p(x, y) \wedge q(x, y) \wedge r(x, y) \Rightarrow \perp}{s(x) \vee q(y, y) \quad p(x, y) \wedge q(x, y) \Rightarrow \perp}}{p(x, x) \Rightarrow s(z)}}{\frac{\frac{q(x, y) \Rightarrow r(x, y) \quad p(x, y) \wedge q(x, y) \wedge r(x, y) \Rightarrow \perp}{s(x) \Rightarrow \exists y q(x, y) \quad p(x, y) \wedge q(x, y) \Rightarrow \perp}}{\forall y p(x, y) \wedge s(x) \Rightarrow \perp}}{p(x, x) \wedge \forall u p(z, u) \Rightarrow \perp}$$

To make the notation more readable, in forward chaining proofs, we do not write the hole context, but only the last derived fact. The forward chaining proofs for $p(x, x) \Rightarrow s(z)$ and $\forall y p(x, y) \wedge s(x) \Rightarrow \perp$ are:

$$\begin{array}{c}
\frac{\perp \vdash s(b)}{r(a, a) \vdash s(b)} \Rightarrow (Ax1) \\
\frac{\frac{\perp \vdash s(b)}{r(a, a) \vdash s(b)} \text{ Inst}}{q(a, a) \vdash s(b)} \text{ Inst} \\
\frac{\frac{\frac{s(b) \vdash s(b)}{s(x) \vdash s(b)} \text{ Inst}}{r(y, y) \vdash s(b)} \text{ Inst}}{q(y, y) \vdash s(b)} \Rightarrow (Ax3) \\
\frac{\frac{\frac{s(b) \vdash s(b)}{s(x) \vdash s(b)} \text{ Inst}}{r(y, y) \vdash s(b)} \text{ Inst}}{q(y, y) \vdash s(b)} \text{ Inst}}{\mathcal{A}\mathcal{X}, p(a, a) \vdash s(b)} \vee
\end{array}
\qquad
\begin{array}{c}
\frac{\perp \vdash \perp}{p(a, b) \vdash \perp} \Rightarrow (Ax1) \\
\frac{\frac{\perp \vdash \perp}{p(a, b) \vdash \perp} \text{ Inst}}{r(a, b) \vdash \perp} \text{ Inst} \\
\frac{\frac{\perp \vdash \perp}{p(a, b) \vdash \perp} \text{ Inst}}{q(a, b) \vdash \perp} \Rightarrow (Ax3) \\
\frac{\frac{\perp \vdash \perp}{p(a, b) \vdash \perp} \text{ Inst}}{q(a, b) \vdash \perp} \text{ Inst}}{\exists y q(a, y) \vdash \perp} \exists \\
\frac{\exists y q(a, y) \vdash \perp}{\mathcal{A}\mathcal{X}, p(a, y), s(a) \vdash \perp} \Rightarrow (Ax2)
\end{array}$$

If we denote the first one by Pr_1 and the second one by Pr_2 , and apply the rules applied in Pr_1 starting from the root $\mathcal{A}\mathcal{X}, p(a, z) \vdash \perp$, we obtain Pr'_1 . Then starting from each leaf of Pr'_1 that did not end in contradiction, we can apply all the rules applied in Pr_2 . The obtained proof is the proof for conjecture:

$$\begin{array}{c}
\frac{\perp \vdash \perp}{p(a, b) \vdash \perp} \Rightarrow (Ax1) \\
\frac{\frac{\perp \vdash \perp}{p(a, b) \vdash \perp} \text{ Inst}}{r(a, b) \vdash \perp} \text{ Inst} \\
\frac{\frac{\perp \vdash \perp}{p(a, b) \vdash \perp} \text{ Inst}}{q(a, b) \vdash \perp} \Rightarrow (Ax3) \\
\frac{\frac{\perp \vdash \perp}{p(a, b) \vdash \perp} \text{ Inst}}{q(a, b) \vdash \perp} \text{ Inst}}{\exists y q(a, y) \vdash \perp} \exists \\
\frac{\exists y q(a, y) \vdash \perp}{s(a) \vdash \perp} \Rightarrow (Ax2) \\
\frac{s(a) \vdash \perp}{s(x) \vdash \perp} \text{ Inst} \\
\frac{s(x) \vdash \perp}{\mathcal{A}\mathcal{X}, p(a, z) \vdash \perp} \vee
\end{array}
\qquad
\begin{array}{c}
\frac{\perp \vdash \perp}{p(a, a) \vdash \perp} \Rightarrow (Ax1) \\
\frac{\frac{\perp \vdash \perp}{p(a, a) \vdash \perp} \text{ Inst}}{r(a, a) \vdash \perp} \text{ Inst} \\
\frac{\frac{\perp \vdash \perp}{p(a, a) \vdash \perp} \text{ Inst}}{q(a, a) \vdash \perp} \text{ Inst} \\
\frac{\frac{\perp \vdash \perp}{p(a, a) \vdash \perp} \text{ Inst}}{r(y, y) \vdash \perp} \text{ Inst} \\
\frac{\frac{\perp \vdash \perp}{p(a, a) \vdash \perp} \text{ Inst}}{r(y, y) \vdash \perp} \text{ Inst}}{q(y, y) \vdash \perp} \Rightarrow (Ax3) \\
\frac{q(y, y) \vdash \perp}{\mathcal{A}\mathcal{X}, p(a, z) \vdash \perp} \vee
\end{array}$$

A corresponding readable (Isar-style) proof would be as follows: *Assume $\forall z p(a, z)$. With (Ax4), it holds $\forall x s(x)$ or $\forall y q(y, y)$. Assume $\forall x s(x)$. From $\forall x s(x)$, it holds $s(a)$. With (Ax2), it holds $\exists y q(a, y)$. From $\exists y q(a, y)$, obtain b such that $q(a, b)$. With (Ax3), it holds $r(a, b)$. From $\forall z p(a, z)$, it holds $p(a, b)$. With (Ax1), this leads to a contradiction. Assume $\forall y q(y, y)$. With (Ax3), it holds $\forall y r(y, y)$. From $\forall y q(y, y)$, it holds $q(a, a)$. From $\forall y r(y, y)$, it holds $r(a, a)$. From $\forall z p(a, z)$, it holds $p(a, a)$. With (Ax1), this leads to a contradiction. All the branches are closed and the conjecture has been proven.*

6 Related Work

There are several proving procedures for coherent logic and similar fragments of FOL, and several corresponding automated theorem provers. To our knowledge, the first CL automated theorem prover was developed in Prolog by Janičić and Kordić [9] and was used for one axiomatization of Euclidean geometry. This prover was later reimplemented in C++ to give a more efficient and generic theorem prover ArgoCLP that produces both natural language proofs and object level proofs in the Isabelle form [18]. Bezem and Coquand developed in Prolog a sound and complete CL prover [2] based on breadth-first search that generates proof objects in Coq. Berghofer and Bezem developed an internal prover for CL in ML to be used within the system Isabelle [19]. Neither of these provers uses backjumps or lemma learning. De Nivelle implemented a theorem prover for logic close to coherent logic, that uses a mechanism for learning lemmas of somewhat restricted form [15]. All of these systems perform only ground reasoning.

Our work is also related to research focused on CDCL-based SAT solvers. Various modifications to the original DPLL procedure have been proposed, both on the high, logical level, and on the lower, algorithmic and implementation

level [5]. Modern SAT solvers have been recently described (with some implementation features omitted) via abstract state transition systems [13,10]. These systems provided a solid ground for rigorous analysis of the SAT solvers and their correctness was formally proved within the system Isabelle [12]. On the operational, practical level, our CL system enables the transfer of SAT algorithms and heuristics (that had a great impact on SAT solving) to CL in some form.

Our system builds on the SAT system presented in Section 2. The differences are primarily due to the first order nature and the form of coherent logic. Use of existential quantifiers results in use of Intro rule which is not present in SAT. Also, dealing with first order formulae results in more complex rules due to the use of substitutions and quantifiers. At this moment we do not include Forget and Restart rules, which can be trivially added, but with them an additional care has to be taken not to jeopardize completeness.

Our work is also related to work on *effectively propositional logic*, also known as EPR, or as the Bernays-Schönfinkel fragment of first-order logic [16].

Another lifting of DPLL procedure, to the clausal fragment of first order logic, is the Model evolution calculus [1]. There are several differences between this system and ours. The first is the underlying logic itself. Working in CL, one can avoid transformation to clausal form when working with coherent theories (like geometry). In our system, the refutation is avoided and forward proofs are used. Skolemization is avoided and existential quantifiers are used. These properties enable generation of readable proofs, close to proofs from mathematical textbooks. Also, in Model evolution calculus, backjump is not treated as a part of the calculus, but as an implementation technique [1].

7 Conclusions and Future Work

In this paper we presented an abstract state transition system for proving validity in coherent logic, but also in a somewhat broader fragment of first order logic. The system is sound and complete: any formula proved by the system is indeed a theorem, and for any input theorem, the system can and will prove it. This also proves the semidecidability of the defined extension of coherent logic. The system is based on a transition system for CDCL-based SAT solving. In contrast to other coherent theorem provers, the reasoning need not to be ground. An important property is that the system allows the generation of formal and human readable forward chaining proofs and we showed how they can be generated.

We are currently developing the implementation that faithfully matches the presented system and expect it to perform well compared to the existing provers. Heuristics that guide applications of the rules should be devised to improve the performance, hopefully in the spirit of heuristics for SAT [14]. Also, the generation of formal (in Isabelle/Isar) and readable object-level proofs will be implemented. We are planning to use the prover for a range of applications, including applications in formalization of mathematics, education, and program synthesis.

References

1. Baumgartner, P., Tinelli, C.: The Model Evolution Calculus as a First-Order DPLL Method. *Artificial Intelligence* 172(4-5) (2008)
2. Bezem, M., Coquand, T.: Automating Coherent Logic. In: Sutcliffe, G., Voronkov, A. (eds.) *LPAR 2005*. LNCS (LNAI), vol. 3835, pp. 246–260. Springer, Heidelberg (2005)
3. M. Bezem. On the Undecidability of Coherent Logic. *Processes, Terms and Cycles* (2005)
4. Bezem, M., Hendriks, D.: On the Mechanization of the Proof of Hessenberg’s Theorem in Coherent Logic. *J. of Automated Reasoning* 40(1) (2008)
5. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): *Handbook of Satisfiability*. IOS Press (2009)
6. Davis, M., Putnam, H.: A Computing Procedure for Quantification Theory. *J. of ACM* 7(3) (1960)
7. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. *Communications of the ACM* 5(7) (1962)
8. Fisher, J., Bezem, M.: Skolem Machines and Geometric Logic. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) *ICTAC 2007*. LNCS, vol. 4711, pp. 201–215. Springer, Heidelberg (2007)
9. Janičić, P., Kordić, S.: EUCLID — the geometry theorem prover. *FILOMAT* 9(3) (1995)
10. Krstić, S., Goel, A.: Architecting Solvers for SAT Modulo Theories: Nelson-Oppen with DPLL. In: Konev, B., Wolter, F. (eds.) *FroCos 2007*. LNCS (LNAI), vol. 4720, pp. 1–27. Springer, Heidelberg (2007)
11. Marić, F.: Formalization and Implementation of Modern SAT Solvers. *J. of Automated Reasoning* 43(1) (2009)
12. Marić, F., Janičić, P.: Formalization of Abstract State Transition Systems for SAT. *Logical Methods in Computer Science* 7(3) (2011)
13. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. of ACM* 53(6) (2006)
14. Nikolić, M., Marić, F., Janičić, P.: Instance-Based Selection of Policies for SAT Solvers. In: Kullmann, O. (ed.) *SAT 2009*. LNCS, vol. 5584, pp. 326–340. Springer, Heidelberg (2009)
15. de Nivelle, H., Meng, J.: Geometric Resolution: A Proof Procedure Based on Finite Model Search. In: Furbach, U., Shankar, N. (eds.) *IJCAR 2006*. LNCS (LNAI), vol. 4130, pp. 303–317. Springer, Heidelberg (2006)
16. Piskrač, R., de Moura, L., Björner, N.: Deciding effectively propositional logic using DPLL and substitution sets. *J. of Automated Reasoning* 44 (2010)
17. Polonsky, A.: *Proofs, Types, and Lambda Calculus*. PhD thesis, University of Bergen (2010)
18. Stojanović, S., Pavlović, V., Janičić, P.: Automated Generation of Formal and Readable Proofs in Geometry Using Coherent Logic. In: Schreck, P., Narboux, J., Richter-Gebert, J. (eds.) *ADG 2010*. LNCS, vol. 6877, Springer, Heidelberg (2011)
19. Wenzel, M.: *Isar - A Generic Interpretative Approach to Readable Formal Proof Documents*. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) *TPHOLs 1999*. LNCS, vol. 1690, pp. 167–183. Springer, Heidelberg (1999)

Speeding Up Cylindrical Algebraic Decomposition by Gröbner Bases*

David J. Wilson, Russell J. Bradford, and James H. Davenport

Department of Computer Science,
University of Bath
Bath BA2 7AY, U.K.

{D.J.Wilson,R.J.Bradford,J.H.Davenport}@bath.ac.uk

Abstract. Gröbner Bases [Buc70] and Cylindrical Algebraic Decomposition [Col75,CMMXY09] are generally thought of as two, rather different, methods of looking at systems of equations and, in the case of Cylindrical Algebraic Decomposition, inequalities. However, even for a mixed system of equalities and inequalities, it is possible to apply Gröbner bases to the (conjoined) equalities before invoking CAD. We see that this is, quite often but not always, a beneficial preconditioning of the CAD problem.

It is also possible to precondition the (conjoined) inequalities with respect to the equalities, and this can also be useful in many cases.

1 Introduction

Solving systems of equations, or equations and inequations (\neq)/inequalities ($>$, $<$) is an old subject. Deciding the truth of, or more generally eliminating quantifiers from, quantified Boolean combinations of such statements, is more recent [Tar51]. We can distinguish many families of methods, even if we restrict attention to the real numbers, or possibly the complex numbers.

$=_G$ The method of Gröbner bases. Here the input is a set $S = \{s_1, \dots, s_k\}$ of polynomials in some polynomial ring $k[x_1, \dots, x_n]$ equipped with a total order \prec on the monomials, and the output is a set $G = \{p_1, \dots, p_l\}$ which is equivalent, in the sense that it generates the same ideal, i.e. $(G) = (S)$, and is simpler, or “surprise-free”, in that the leading monomial with respect to \prec (denoted lm_\prec) behaviour is explicit, $(\text{lm}_\prec(G)) = (\text{lm}_\prec((G)))$. Then the solutions of G are those of S , i.e.

$$\{\mathbf{x} : p_1(\mathbf{x}) = 0 \wedge p_2(\mathbf{x}) = 0 \wedge \dots \wedge p_l(\mathbf{x}) = 0\}. \quad (1)$$

* The examples used in this paper are available in [Wil12]. This work was partially supported by the U.K.’s EPSRC under grant number EP/J003247/1.

¹ We have concentrated on purely lexicographical orders, since these seem to be the most useful to us.

$=_{\Delta}$ The method of triangular decomposition via regular chains [ALMM99,MM05]. Here the output is a set of regular chains of polynomials

$$\{(p_{1,1}, p_{1,2}, \dots), (p_{2,1}, p_{2,2}, \dots), \dots\}, \tag{2}$$

and the solution is the union of the set of regular zeros of these regular chains:

$$\left\{ \mathbf{x} : p_{1,1}(\mathbf{x}) = p_{1,2}(\mathbf{x}) = \dots = 0 \wedge \left(\prod_i \text{init}(p_{1,i}) \right) (\mathbf{x}) \neq 0 \right\} \cup \dots \tag{3}$$

$<_{\text{Col}}$ The method of Cylindrical (semi-)Algebraic Decomposition for real closed fields, computed via repeated projection to \mathbf{R}^1 and repeated lifting [Col75, and many improvements].

$=_{\text{Col}}$ The previous case restricted to equality.

$<_{\Delta\mathbf{R}}$ The method of Cylindrical (semi-)Algebraic Decomposition for real closed fields via triangular decomposition [CMMXY09].

$\neq_{\Delta\mathbf{C}}$ The method of Cylindrical Decomposition over the complexes via triangular decomposition, which was introduced in [CMMXY09] as a stepping-stone to the previous method, but which probably has independent interest.

\cancel{A}_{CH} Quantifier Elimination by partial (i.e. taking account of the Boolean structure and quantifier structure) Cylindrical Algebraic Decomposition [CH91].

Others such as Weispfenning’s Virtual Term Substitution [Bro05, is a readable introduction], or Tarski’s original method [Tar51].

Conversely instead of asking for solutions \mathbf{x} to $\exists \mathbf{x} f_1(\mathbf{x}) \geq 0 \wedge \dots$, we may use a Positivstellensatz to show that no such \mathbf{x} exist, as in [PQR09]. We do not discuss this direction further here.

It should be noted that both $<_{\text{Col}}$ and $<_{\Delta\mathbf{R}}/\neq_{\Delta\mathbf{C}}$ (but not \cancel{A}_{CH}) have the drawback that the Cylindrical Algebraic Decomposition produces decompositions for, not only the question posed, e.g. $\forall y \exists z p(x, y, z) = 0 \wedge q(x, y, z) = 0 \wedge r(x, y, z) > 0$, but also *all* other questions involving the same polynomials, provided the quantifiers are over variables in the same order, e.g. $\exists y \forall z p(x, y, z) < 0 \vee (q(x, y, z) > 0 \wedge r(x, y, z) = 0)$.

This paper asks the question: “can these methods usefully be combined?” The combinations we are thinking about are those of conjunction: Can the fact that B is in the context of $a_1 = 0 \wedge \dots \wedge a_k = 0 \wedge B$ be used to simplify B ? In particular, we look at the use of Gröbner base methods to simplify the equalities in the conjunction *and* to simplify the inequalities in the light of the equalities.

Technical Note: All computations ($=_G$, $<_{\Delta\mathbf{R}}$ and $\neq_{\Delta\mathbf{C}}$) were performed in Maple 16 β on a 3.1GHz Intel processor, except for the $<_{\text{Col}}$, $=_{\text{Col}}$ and \cancel{A}_{CH} ones, which were performed on a 2.83GHz Intel processor with QEPCAD B version 1.65 [Bro03]. Times for a hybrid calculation, e.g. $=_G/<_{\text{Col}}$, are either quoted as the total time or a decomposition $a + b = c$ where a is the time (in milliseconds) for $=_G$, b for $<_{\text{Col}}$, and c is the sum. We have run QEPCAD in three modes:

1. on the problem as given in [BH91], implementing \bar{A}_{CH} ;
2. as above but with the `full-cad` option to ignore the Boolean structure of the expression;
3. with no quantifiers stated, and the `full-cad` option, implementing $<_{Col}$.

2 Examples in This Paper

2.1 [BH91]

This paper has a variety of examples for \bar{A}_{CH} , all of a form to which $=_G$ is applicable.

2.2 [CMMXY09]

This paper has a variety of examples for $<_{\Delta R}$. We chose some of those to which $=_G$ is applicable.

2.3 Two Spheres and a Cylinder

Let the following be spheres in \mathbf{R}^3 :

$$\begin{aligned} S_1 : & \quad (x - 1)^2 + y^2 + z^2 - 3; \\ S_2 : & \quad (x + 1)^2 + y^2 + z^2 - 3; \\ S_3 : & \quad (x - 1)^2 + \left(y - \frac{1}{2}\right)^2 + z^2 - 3; \\ S_4 : & \quad \left(x + 1\right)^2 + \left(y + \frac{2}{3}\right)^2 + \left(z + \frac{3}{4}\right)^2 - 3. \end{aligned}$$

Denote the infinite cylinder centred on the z -axis with radius 1 by C , so that the equation defining the cylinder is:

$$C : \quad x^2 + y^2 - 1.$$

Now we investigate intersecting pairs of spheres (roughly increasing in CAD ‘difficulty’) under conditions based on the cylinder. We assume the spheres’ equation will always be required to equal 0 but make no assumptions on the condition on the cylinder. That is, we wish to solve the problem:

$$S_i = 0 \wedge S_{i+1} = 0 \wedge C * 0 \quad * \in \{=, \neq, <, >, \leq, \geq\}, i = 1, 2, 3. \quad (4)$$

We use the underlying variable ordering² (z, y, x) .

² This is the QEPCAD notation, meaning that we will project from (z, y, x) -space to (z, y) -space to (z) -space. We therefore end up with polynomials in z alone, so this is equivalent to a purely lexicographical Gröbner base with $z \prec y \prec x$, i.e. `plex([z,y,x])` in Maple: $=_G$ is used to indicate Gröbner bases with this (compatible) ordering. The CAD package in Maple [CMMXY09] requires `PolynomialRing([x,y,z])` to achieve the same effect as QEPCAD’s (z, y, x) . $=_{GR}$ denotes the reverse `plex` order.

3 Prior Art

Needless to say, we are not the first to have had this idea.

3.1 Buchberger–Hong

[BH91] considers the case of $=_G$ ([BGK85] re-implemented in C) applied to $<_{Col}$ (an early version of [CH91] re-implemented in C), i.e., rather than computing a CAD for the zeros of a system of equations E (i.e. $e_1 = 0 \wedge e_2 = 0 \wedge \dots$) and inequalities F , compute it for G , a (purely lexicographical) Gröbner base for E , and F . They generally found a very substantial speed-up in the total computation time, e.g. “Solotareff A”³

$$\exists x \exists y \quad 3x^2 - 2x - a = x^3 - x^2 - ax - 2b + a - 2 = \tag{5}$$

$$3y^2 - 2y - a = y^3 - y^2 - ay - 2b + a - 2 = 0 \wedge \tag{6}$$

$$4a \in [1, 7] \wedge 4b \in [-3, 3] \wedge x \in [-1, 0] \wedge y \in [0, 1] \tag{7}$$

(with the variable ordering (b, a, x, y)) took them 11500 ms for \bar{A}_{CH} , but 717 for $=_G$, and 117 for \bar{A}_{CH} applied to the result, a total of 834 ms, or a 13-fold speed-up. “Solotareff B” is the same problem but with (a, b, x, y) as the variable ordering, and here the \bar{A}_{CH} time was again greatly reduced, but the $=_G$ time was excessive. Of course, there have been substantial improvements in the implementation of all these algorithms since [BH91] was published, and Table 2 shows that the $=_G$ time is now less than 1/3 of the \bar{A}_{CH} time. We choose rather to focus on the number of cells generated, which is closely connected to the $=_{Col}$ time, and also affects the time taken to make use of the output. The cell counts are shown in Table 1.

Table 1. Cell counts for Solotareff

		Ordering A		Ordering B	
		$<_{Col} =_G / <_{Col}$	$<_{Col}$	$<_{Col} =_G / <_{Col}$	$<_{Col}$
(5-7)	Partial	153	63	375	41
	Full	349	625	1063	237
(5-6)	Partial	29	15	97	17
	Full	29	33	97	17

More reruns of [BH91] are given in Table 2. We see that, with today’s technology, the conclusion of [BH91], viz. that $=_G$ generally improves \bar{A}_{CH} for the class of problems to which it is applicable, is still generally valid, but the details differ: notably the Gröbner base time is generally insignificant today.

³ There are various problems labelled “Solotareff”: for a description of this class see [Wil12] and the links therein.

Table 2. [BH91] with today’s technology

	$\bar{\Delta}_{CH}$		$=_G/\bar{\Delta}_{CH}$			$\bar{\Delta}_{CH}/\text{full-cad}$		$=_G/\bar{\Delta}_{CH}/\text{full-cad}$		
	Time	Cells	Time	Cells	Cells	Time	Cells	Time	Cells	Cells
I A	190	503	22+72=	94	23	188	503	22+73=	95	51
I B	199	369	21+74=	95	17	191	369	21+75=	96	33
R A	85	1	24+73=	97	1	86	1	24+71=	95	1
R B	129	1	24+72=	96	1	125	1	24+72=	96	1
E A	297	621	25+134=	159	621	576	11139	25+394=	419	11139
E B	Error	?	50+?=	Error	?	Error	?	50+?=	Error	?
S A	89	153	22+72=	94	63	199	349	22+185=	207	625
S B	113	375	23+75=	98	41	228	1063	23+180=	203	237
C A	133	19	42+?=	Error	?	235	19	42+?=	Error	?
C B	Error	?	132+?=	Error	?	Error	?	132+?=	Error	?

Table 3. [BH91] Examples for full CADs

	$=_{CoI}$		$=_G/=_{CoI}$			$<_{\Delta R}$		$=_G/<_{\Delta R}$	
	Time	Cells	Time	Cells	Cells	Time	Cells	Time	Cells
I A	236	3723	22+77=	99	273	29426	3763	2470	273
I B	212	3001	21+76=	97	189	36262	2795	1482	189
R A	150	2101	24+86=	110	105	17355	1267	570	165
R B	21091	7119	24+80=	104	141	356670	7119	470	141
E A*	7390	114541	25+3189=	3214	53559	262623	28557	62496	14439
E B*	Error	?	50+?=	Error	?	> 1000s	?	> 1000s	?
S A*	115	1751	22+82=	104	297	16014	1751	2025	297
S B*	253	6091	23+82=	105	243	43439	6091	1647	243
C A*	820	8387	42+?=	Error	?	216028	7895	> 1000s	?
C B*	Error	?	132+?=	Error	?	> 1000s	?	> 1000s	?

* indicates that the linear inequalities have been omitted in this version.

There is one point which is not explicit in [BH91]. As the computation of Gröbner bases in one variable is just equivalent to Euclid’s algorithm, i.e. Gaussian elimination in Sylvester’s matrix, Gröbner base computations which are not genuinely multi-variate do not affect the set of resultants etc. generated in $<_{CoI}$, and hence are of limited use in the projection phase. They might still reduce the work done in the lifting phase, of course.

Table 3 re-runs the examples of [BH91], but asking for complete cylindrical algebraic decompositions, and hence we can compare $<_{CoI}$ with $<_{\Delta R}$ legitimately. Given that the algorithms are fundamentally different, the similarities in cell counts are striking. The differences in cell counts (where present) reflect differences in the cylindrical algebraic decompositions for the same input problem.

3.2 Phisanbut

Phisanbut [Phi11], considering branch cuts in the complex plane, observed that $g = 0 \wedge f > 0$ could be reduced to $g = 0 \wedge \text{prem}(f, g) > 0$ under suitable conditions, where prem denotes the pseudo-remainder operation. More precisely, if f and g are regarded as polynomials in the main variable x , of degrees d and e respectively, then $\text{prem}(f, g) = \text{rem}(c^{d-e+1}f, g)$, where c is the leading coefficient of g . When $g = 0$ and $c > 0$, or when $d - e + 1$ is even, $\text{prem}(f, g)$ has the same sign as f . Unfortunately c might have variable sign, and $d - e + 1$ might be odd, so define $\text{pprecond}(f, g) = \text{rem}(c^{(d-e+1)^*}f, g)$, where n^* is n if n is even and $n + 1$ if n is odd. Maple also defines $\text{sprem}(f, g) = \text{rem}(c^m f, g)$, where m is the smallest integer such that the division is exact, and by analogy we have $\text{sprecond}(f, g) = \text{rem}(c^{m^*} f, g)$. Note that $\text{sprecond}(f, g) = \text{pprecond}(f, g)$ or a strict divisor of it, i.e. sprecond is never worse. She generally, but not always, saw [Phi11, Tables 8.13, 8.14] a significant decrease in the number of cells, and the time taken to compute sprecond was minimal.

4 Further Developments

4.1 $=_G$ with $<_{\Delta R}$

It would seem natural to apply $=_G$ to $<_{\Delta R}$, as [BH91] did to \bar{A}_{CH} . The results are in Table 3, and show a speed-up in all instances except the Collision problems. We also note the substantial speed advantage enjoyed by $<_{Col}$, and this is a subject for further study.

4.2 $=_G$ with $\neq_{\Delta C}$

We can also mix $=_G$ with $\neq_{\Delta C}$, and these results are shown in Table 4, which also compares $\neq_{\Delta C}$ with $<_{\Delta R}$. $<_{\Delta R}$ involves doing $\neq_{\Delta C}$ first, and then running the `MakeSemiAlgebraic` algorithm from [CMMXY09]. For these examples, the `MakeSemiAlgebraic` step is the most expensive initially, but often not after we apply $=_G$.

4.3 $=_G$ with Inequalities in $<_{\Delta R}$

Having reduced the equalities to a Gröbner base G , it is now possible to reduce the inequalities by G , since adding/subtracting a multiple of an element of G is adding/subtracting 0. We can reduce with respect to the main variable, denoted $=_G / \rightarrow_x^G$, with respect to secondary variables, denoted $=_G / \rightarrow_y^G$, or with respect to all variables (Maple’s `NormalForm`), denoted $=_G / \overset{*}{\rightarrow}^G$. If we compare tables 6 and 7 we see that the number of cells produced is the same across the two methods.

Table 4. Timings for [BH91] Examples: $\langle \Delta_R / \neq \Delta_C$

	$\neq \Delta_C$ Time	$\langle \Delta_R$ Time	Ratio	$=_G / \neq \Delta_C$ Time	$=_G / \langle \Delta_R$ Time	Ratio
Intersection A	5691	29426	4.17	1168	2470	1.11
Intersection B	5584	36262	5.49	886	1482	0.67
Random A	4614	17355	2.76	310	570	0.84
Random B	67343	356670	4.30	318	470	0.48
Ellipse A*	85425	262623	2.07	27916	62496	1.24
Ellipse B*	441245	> 1000s	-	> 1000s	> 1000s	-
Solotareff A*	6666	16014	1.40	1760	2025	0.15
Solotareff B*	9536	43439	3.56	1404	1647	0.17
Collision A*	41085	216028	4.26	> 1000s	> 1000s	-
Collision B*	> 1000s	> 1000s	-	> 1000s	> 1000s	-

“Ratio” = $(\langle \Delta_R - \neq \Delta_C) / \neq \Delta_C$, i.e. the relative cost of MakeSemiAlgebraic.

Table 5. Examples from [CMMXY09]

	$\langle \Delta_R$ Time Cells		$=_G / \langle \Delta_R$ Time Cells		Ratio Time Cells		
Cyclic-3	3136	381	20 + 245 =	265	21	11.83	18.14
Cyclic-4	> 1000s	?	64 + 5813 =	5877	621	?	?
2	2249	895	22 + 1845 =	1867	579	1.20	1.55
4	3225	421	24 + 19738 =	19762	1481	0.16	0.28
6	363	41	20 + 918 =	938	89	0.39	0.46
7	3667	895	26 + 6537 =	6563	1211	0.56	0.74
8	3216	365	21 + 174 =	195	51	16.49	7.16
13	14342	4949	18 + 220 =	238	81	60.26	61.10
14	334860	27551	21 + 971 =	992	423	337.56	65.13

Table 6. Spheres and Cylinders: $\langle \Delta_R$

	$\langle \Delta_R$ Time Cells		$=_G / \langle \Delta_R$ Time Cells		$=_G / \rightarrow_y^G / \langle \Delta_R$ Time Cells		$=_G / \rightarrow_x^G / \langle \Delta_R$ Time Cells		$=_G / \rightarrow^{*G} / \langle \Delta_R$ Time Cells	
S_1, S_2, C	9830	1073	1057	267	394	91	528	183	298	99
S_2, S_3, C	187048	12097	5880	1299	3171	627	2149	517	506	213
S_3, S_4, C	247458	11957	8164	1359	9177	1123	5476	881	590	213

Table 7. Spheres and Cylinders: $\langle Col$

	$\langle Col$ Time Cells		$=_G / \langle Col$ Time Cells		$=_G / \rightarrow^{*G} / \langle Col$ Time Cells			
S_1, S_2, C	30	1073	23 + 8 =	31	267	24 + 4 =	28	99
S_2, S_3, C	763	12097	27 + 36 =	63	1299	28 + 13 =	41	213
S_3, S_4, C	1760	11957	28 + 37 =	65	1359	29 + 14 =	43	213

5 Choice of Method

Suppose we are given a problem, which we may formulate as

$$\text{quantified variables } e_1 = 0 \wedge \cdots \wedge e_k = 0 \wedge B(f_1, \dots, f_l), \quad (8)$$

where B is a Boolean combination of conditions $= 0, \neq 0, < 0$ etc. on some polynomials f_j , then we may be able, by applying Gröbner techniques to the e_j , producing $e_j^{(i)}$, and then reducing the f_j , to produce various alternative formulations

$$\text{quantified variables } e_1^{(i)} = 0 \wedge \cdots \wedge e_{k^{(i)}}^{(i)} = 0 \wedge B(f_1^{(i)}, \dots, f_l^{(i)}), \quad (8^{(i)})$$

and each of these may have several variable orderings compatible with the constraints implied by the quantification (if any). Which should we choose? Of course, in the presence of arbitrary parallelism, we can start them all, and accept the first to finish, but we may wish to be less extravagant.

In the contexts of \mathcal{A}_{CH} (strictly speaking, the REDLOG implementation), and where the only choice was in the variable order, this question was considered by [DSS04]. Retrospectively, there are two measures for the difficulty of a CAD computation: the time taken and the number of cells produced. For a given \mathcal{A}_{CH} problem, they observed that two are usually correlated for different formulations, and we observe the same here for $<_{\Delta\mathbf{R}}$ — see our tables. However, we would like a measure that could be calculated in advance, rather than retrospectively.

The processes of [Col75][CH91] starts with a set A_n of polynomials in n (ordered) variables x_1, \dots, x_n , and

1. repeatedly project A_i into A_{i-1} in one fewer variable, until A_1 has only one variable,
- * (denote the set $\{A_n, \dots, A_1\}$ by $A(x_1, \dots, x_n)$)
2. isolate the roots of these polynomials to get a decomposition of \mathbf{R}^1 ,
3. repeatedly lift the decomposition until we get a (partial for [CH91]) cylindrical algebraic decomposition of \mathbf{R}^n .

The third step is, both theoretically and practically, by far the most expensive. Hence the question arises: what can we measure at the end of step 1, i.e. depending on A only, which is well-correlated with the final cost? Three things come to mind.

$$\text{card}(A(x_1, \dots, x_n)) = \sum_{i=1}^n |A_i|.$$

$$\text{td}(A(x_1, \dots, x_n)) = \sum_{i=1}^n \sum_{p_{i,j} \in A_i} \text{td}(p_{i,j}) \text{ where } \text{td} \text{ denotes total degree.}$$

$$\text{sotd}(A(x_1, \dots, x_n)) = \sum_{i=1}^n \sum_{p_{i,j} \in A_i} \sum_{\text{monomials } m \text{ of } p_{i,j}} \text{td}(m).$$

[DSS04] discard td , observing that td and sotd are highly correlated and sotd “has the advantage of favouring sparse polynomials”. They then observe that $\text{sotd}(A(x_1, \dots, x_n))$ is significantly more correlated with the retrospective measures for any given problem than card . This gives a first algorithm for deciding how to project: for all admissible (i.e. compatible with the quantifier structure, if

any) permutations π of (x_1, \dots, x_n) , compute $A(x_{\pi(1)}, \dots, x_{\pi(n)})$, and choose the one with the least `sotd` value. The drawback of this is that it requires potentially $(n-1)n!$ projection operations. They show that (at least on their examples) this always produces a good projection order, and frequently the optimal.

Table 8. Spheres and Cylinders: $<_{\Delta R}$ — choice of orderings

		$<_{\Delta R}$	$=_G / <_{\Delta R}$	$=_G / \overset{*}{\rightarrow}^G / <_{\Delta R}$
		Time Cells	Time Cells	Time Cells
S_1, S_2, C	C	8654 1073	905 267	270 99
	R		902 267	453 183
S_2, S_3, C	C	189202 12097	5911 1299	499 213
	R		18941 2639	5307 859
S_3, S_4, C	C	248340 11957	8159 1359	580 213
	R		160171 9091	196714 11203

They therefore propose a *greedy algorithm*, where for all permissible choices of the first variable to be projected, we compute `sotd`(A_{n-1}), and choose the variable which gives the least value. Having fixed this as the first variable to project, for all permissible choices of the second variable to be projected, we compute `sotd`(A_{n-2}), and choose the variable which gives the least value, and so on. Hence, assuming all projection orders are possible, the **number** of projections done is $n + (n-1) + \dots = O(n^2)$ rather than $n!$. It is currently an open question whether the **cost** of projections behaves similarly.

We proposed taking this idea still further, and suggested that, for several different formulations A_n, B_n, \dots of a problem, we should compute `sotd`(A_n), `sotd`(B_n), \dots and take the formulation that yields the lowest `sotd`. We observed, however, that neither `td` nor `sotd` are good predictors in Table [11](#), despite seeming useful in Table [10](#).

6 The Metric TNoI

When we apply Gröbner techniques to a set of equations (either by calculating a basis or a normal form) we are, in some sense, trying to simplify the set of equations. In a zero-dimensional ideal, as shown in the Gianni-Kalkbrener Theorem [\[Gia89, Kal89\]](#), a purely lexicographic Gröbner basis has a very distinct, triangular structure.

With this in mind we thought it may be of some use to consider the number of variables present in a certain problem and so defined the following quantity, TNoI, which stand for “Total Number of Indeterminates”:

$$\text{TNoI}(F) = \sum_{f \in F} \text{NoI}(f), \tag{9}$$

where `NoI`(f) is the number of indeterminates present in a polynomial f .

Table 9. [BH91]: effect of orderings $=_{GC}$ versus $=_{GR}$

		$<_{\Delta R}$		$=_G / <_{\Delta R}$	
		Time Cells		Time Cells	
Intersection A	C	29426	3763	2470	273
	R			> 1000s	?
Intersection B	C	36262	2795	1482	189
	R			> 1000s	?
Random A	C	17355	1219	570	165
	R			> 1000s	?
Random B	C	356670	7119	470	141
	R			> 1000s	?
Ellipse A*	C	262623	28557	62496	14439
	R			271726	29939
Ellipse B*	C	> 1000s	?	> 1000s	?
	R			> 1000s	?
Solotareff A*	C	16014	1751	2025	297
	R			> 1000s	?
Solotareff B*	C	43439	6091	1647	243
	R			> 1000s	?
Collision A*	C	216028	7895	> 1000s	?
	R			> 1000s	?
Collision B*	C	> 1000s	?	> 1000s	?
	R			> 1000s	?

We note that $=_{GR}$ is definitely worse than $=_{GC}$.

Table 10. Spheres and Cylinders: $<_{\Delta R}$ —degrees

	$<_{\Delta R}$		$=_G / <_{\Delta R}$			$=_G / \overset{G}{\rightarrow} / <_{\Delta R}$		
	degrees	Time Cells	degrees	Time Cells	degrees	Time Cells	degrees	Time Cells
S_1, S_2, C	6 / 18	8654 1073	5 / 9	905 267	5 / 7	270 99		
S_2, S_3, C	6 / 19	189202 12097	5 / 11	5911 1299	5 / 10	499 213		
S_3, S_4, C	6 / 21	248340 11957	5 / 15	8159 1359	5 / 15	580 213		

'degrees' is $\text{td}(A_n) / \text{sotd}(A_n)$.

Table 11. [BH91]: degrees

	$<_{\Delta R}$			$=_G / <_{\Delta R}$		
	degrees	Time Cells		degrees	Time Cells	
Intersection A	6 / 14	29426	3763	17 / 50	2470	273
Intersection B	6 / 14	36262	2795	15 / 41	1482	189
Random A	9 / 16	17355	1219	19 / 68	570	165
Random B	9 / 16	356670	7119	19 / 73	470	141
Ellipse A*	6 / 24	262623	28557	6 / 26	62496	14439
Ellipse B*	6 / 24	> 1000s	?	25 / 253	> 1000s	?
Solotareff A*	10 / 25	16014	1751	10 / 28	2025	297
Solotareff B*	10 / 25	43439	6091	21 / 69	1647	243
Collision A*	6 / 23	216028	7895	27 / 251	> 1000s	?
Collision B*	6 / 23	> 1000s	?	36 / 875	> 1000s	?

6.1 TNoI Data

The results of calculating such a quantity are given in Table 8, Table 9 and Table 10, showing a promising correlation to whether our preconditioning (with compatible ordering) is beneficial or not. In particular we note the following points:

- In every example where preconditioning reduces TNoI (15 cases) there is a significant reduction in timing (a decrease factor ranging from 4.20 to 757.26) and number of cells produced (a decrease factor ranging from 1.98 to 65.13).
- When preconditioning increases TNoI (7 cases) then generally there is an increase in time (an increase factor ranging from 1.79 to 6.13) and the number of cells created (an increase factor ranging from 1.35 to 3.52) or the problem remains infeasible. There is one ‘false positive’ result ([CMMXY09](#), Example 2]) where there is an increase in TNoI but a slight improvement in the time (a decrease factor of 1.20) and cells produced (a decrease factor of 1.55).
- TNoI alone does not measure the abstract difficulty of the calculations: Intersection A has a higher TNoI than Ellipse A yet the latter takes 25 times longer and produces over 50 times as many cells. We have only shown how to use it to compare variants of the same problem.

As mentioned above, calculating TNoI alone is not of a huge use, and even considering the difference or ratio does little to predict the degree of improvement to expect. However, if we take the logarithm of the ratio (equivalently the difference of the logarithms) of TNoI and compare to the time or number of cells we get some interesting results.

Plotting these quantities against each other certainly suggested there was a positive correlation. Recall that the sample correlation coefficient is defined as

$$r_{X,Y} = \frac{\sum_{i=1}^n (X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^n (X_i - \bar{X})^2} \sqrt{\sum_{i=1}^n (Y_i - \bar{Y})^2}} \quad (10)$$

and is a number between -1 and 1 that indicates how correlated data is. A sample coefficient of 1 indicates perfect positive correlation and a coefficient of -1 indicates perfect negative correlation. Although we are only working with a small bank of data (22 examples) and partially incomplete data (timings of > 1000s were replaced by 10000 seconds and unknown cell numbers were replaced by 100000 to allow for coefficient calculation) there were still promising results.

Let S be the polynomial input, \mathcal{D}_S its corresponding CAD, t_S the time taken to calculate \mathcal{D}_S and c_S the number of cells in \mathcal{D}_S . Let G be the Gröbner basis calculated with respect to the compatible ordering and define \mathcal{D}_G , t_G and c_G in a similar fashion. With the data set we obtained the sample correlation coefficients were as follows:

- comparing $\log(\text{TNoI}(S)) - \log(\text{TNoI}(G))$ with $\log(t_S) - \log(t_G)$ gives a sample coefficient $r = 0.821$ which indicates strong correlation (for our limited sample set).

- comparing $\log(\text{TNoI}(S)) - \log(\text{TNoI}(G))$ with $\log(c_S) - \log(c_G)$ gives a sample coefficient $r = 0.829$ which again indicate a strong correlation (for our limited sample set).

Of course correlation does not imply causation, especially with a relatively small data set, so let us look more deeply at what TNoI is measuring.

Table 12. TNoI for Spheres

	$<_{\Delta R}$			$=_G / <_{\Delta R}$			$=_G / \xrightarrow{*} G / <_{\Delta R}$		
	TNoI	Time	Cells	TNoI	Time	Cells	TNoI	Time	Cells
S_1, S_2, C	8	8654	1073	5	905	267	4	270	99
S_2, S_3, C	8	189202	12097	6	5911	1299	6	499	213
S_3, S_4, C	8	248340	11957	7	8159	1359	7	580	213

Table 13. TNoI for [\[BH91\]](#)

	$<_{\Delta R}$			$=_G / <_{\Delta R}$		
	TNoI	Time	Cells	TNoI	Time	Cells
Intersection A	8	29426	3763	7	2470	273
Intersection B	8	36262	2795	7	1482	189
Random A	9	17355	1219	5	570	165
Random B	9	356670	7119	5	471	141
Ellipse A*	7	262623	28557	6	62496	14439
Ellipse B*	7	> 1000s	?	21	> 1000s	?
Solotareff A*	9	16014	1751	8	2025	297
Solotareff B*	9	43439	6091	7	1647	243
Collision A*	7	216028	7895	18	> 1000s	?
Collision B*	7	> 1000s	?	22	> 1000s	?

6.2 What Is TNoI Measuring?

Consider what causes TNoI to decrease. Let S be a set of polynomials in variables x_1, \dots, x_n ordered $x_1 < x_2 < \dots < x_n$. The following are three possible reasons for a decrease in TNoI:

1. The number of polynomials in a specific set of variables, $\{x_{i_1}, \dots, x_{i_l}\}$, is decreased. If x_k is the most important variable then reducing the number of these polynomials will simplify the decomposition in the (x_1, \dots, x_k) -plane. This will simplify the overall CAD, reducing the number of cells produced and hence the time taken to calculate the decomposition.
2. At least one variable is eliminated from a polynomial. If the variable x_k is eliminated from a polynomial p then the decomposition based around p will be greatly simplified. This will again simplify the overall CAD, reducing the number of cells produced and hence the time taken to calculate the decomposition.

Table 14. TNoI for [CMMXY09](#)

	$<_{\Delta R}$			$=_G / <_{\Delta R}$		
	TNoI	Time	Cells	TNoI	Time	Cells
Cyclic-3	9	3136	381	6	20 + 245 =	265 21
Cyclic-4	16	> 1000s	?	6	64 + 5813 =	5877 621
2	7	2249	895	14	22 + 1845 =	1867 579
4	6	3225	421	11	24 + 19738 =	19762 1481
6	4	363	41	5	20 + 918 =	938 89
7	8	3667	895	22	26 + 6537 =	6563 1211
8	6	3216	365	5	21 + 174 =	195 51
13	9	14342	4949	4	18 + 220 =	238 81
14	11	334860	27551	9	21 + 971 =	992 423

3. A polynomial in a large number of variables, say k , is replaced by j polynomials each with n_i variables such that $\sum n_i < k$. Intuitively this would increase the *number* of discriminants and resultants calculated, be it in the projection phase of $<_{Col}$ or in $\neq_{\Delta C}$, but the results appear in lower levels of the projection tree, and this effect is more potent than the apparent increase in the number of discriminants and resultants. We have yet to build a good model of this, though.

Obviously, in general, a combination of these factors will be the reason for the decrease in TNoI. Also, there may be opposing increases in TNoI, which presumably explains why the ‘false positive’ of [CMMXY09](#), Example 2] shows an increase in TNoI but an improvement in the CAD efficiency.

7 Conclusions

- For both $<_{Col}$ and $<_{\Delta R}$ and $\neq_{\Delta C}$, pre-conditioning the equations (where applicable) by means of a Gröbner calculation is often well worth doing.
- Gröbner reduction of inequalities with respect to equalities has never, in our examples, made things worse.
- *A priori*, it can be quite difficult to see which combinations of Gröbner base and Gröbner reduction will be best, but the Gröbner side is generally cheap⁴.
- We therefore have multiple equivalent formulations of a given problem. We have investigated the metrics of [DSS04](#), but have concluded that, at the level of choice of formulation, TNoI is a better predictor. It does not help for predicting the best ordering of variables, for which [DSS04](#) or the Brown heuristic [Bro04](#) are appropriate. Phisanbut [Phi11](#), Chapter 8] found the Brown heuristic sufficiently good, and simpler to compute.

⁴ This is a significant change from [BH91](#), who had examples where the Gröbner calculations was much more expensive than the Cylindrical Algebraic Decomposition.

- In Section 3.2 we saw how $g = 0 \wedge f > 0$ could be reduced to $g = 0 \wedge \text{sprecond}(f, g) > 0$. In principle, given $s_1 = 0 \wedge \dots \wedge s_k = 0 \wedge f > 0$, after computing a Gröbner base G for the s_i , we could attempt a more general reduction of f by G . Pure NormalForm reduction has proved useful (Tables 6, 7), but we do not have enough good examples to measure the utility of a more general pseudoremainder-like reduction.

References

- ALMM99. Aubry, P., Lazard, D., Moreno Maza, M.: On the Theories of Triangular Sets. *J. Symbolic Comp.* 28, 105–124 (1999)
- BGK85. Böge, W., Gebauer, R., Kredel, H.: Gröbner Bases Using SAC2. In: Caviness, B.F. (ed.) *ISSAC 1985 and EUROCAL 1985*. LNCS, vol. 204, pp. 272–274. Springer, Heidelberg (1985)
- BH91. Buchberger, B., Hong, H.: Speeding-up Quantifier Elimination by Gröbner Bases. Technical Report 91-06 (1991)
- Bro03. Brown, C.W.: QEPCAD B: a program for computing with semi-algebraic sets using CADs. *ACM SIGSAM Bulletin* 37(4), 97–108 (2003)
- Bro04. Brown, C.W.: Tutorial handout (2004), <http://www.cs.usna.edu/~wcbrown/research/ISSAC04/handout.pdf>
- Bro05. Brown, C.W.: SLFQ — simplifying large formulas with QEPCAD B (2005), <http://www.cs.usna.edu/~qepcad/SLFQ/Home.html>
- Buc70. Buchberger, B.: Ein algorithmisches Kriterium für die Lösbarkeit eines algebraischen Gleichungssystem (English translation in [Buc98]). *Aequationes Mathematicae* 4, 374–383 (1970)
- Buc98. Buchberger, B.: An Algorithmic Criterion for the Solvability of a System of Algebraic Equations. In: *Gröbner Bases and Applications*, pp. 535–545 (1998)
- CH91. Collins, G.E., Hong, H.: Partial Cylindrical Algebraic Decomposition for Quantifier Elimination. *J. Symbolic Comp.* 12, 299–328 (1991)
- CMMXY09. Chen, C., Moreno Maza, M., Xia, B., Yang, L.: Computing Cylindrical Algebraic Decomposition via Triangular Decomposition. In: May, J. (ed.) *Proceedings ISSAC 2009*, pp. 95–102 (2009)
- Col75. Collins, G.E.: Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition. In: *Proceedings 2nd GI Conference Automata Theory & Formal Languages*, pp. 134–183 (1975)
- DSS04. Dolzmann, A., Seidl, A., Sturm, T.: Efficient Projection Orders for CAD. In: Gutierrez, J. (ed.) *Proceedings ISSAC 2004*, pp. 111–118 (2004)
- Gia89. Gianni, P.: Properties of Gröbner Bases Under Specializations. In: Davenport, J.H. (ed.) *ISSAC 1987 and EUROCAL 1987*. LNCS, vol. 378, pp. 293–297. Springer, Heidelberg (1989)
- Kal89. Kalkbrener, M.: Solving Systems of Algebraic Equations by Using Gröbner Bases. In: Davenport, J.H. (ed.) *ISSAC 1987 and EUROCAL 1987*. LNCS, vol. 378, pp. 282–292. Springer, Heidelberg (1989)
- MM05. Moreno Maza, M.: On Triangular Decompositions of Algebraic Varieties (2005), <http://www.csd.uwo.ca/~moreno/Publications/M3-MEGA-2005.pdf>
- Phi11. Phisanbut, N.: Practical Simplification of Elementary Functions using Cylindrical Algebraic Decomposition. PhD thesis, University of Bath (2011)

- PQR09. Platzer, A., Quesel, J.-D., Rümmer, P.: Real World Verification. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 485–501. Springer, Heidelberg (2009)
- Tar51. Tarski, A.: A Decision Method for Elementary Algebra and Geometry, 2nd edn. Univ. Cal. Press (1951)
- Wil12. Wilson, D.J.: Real Geometry and Connectness via Triangular Description: CAD Example Bank (2012), <http://opus.bath.ac.uk/29503>

A System for Axiomatic Programming

Gabriel Dos Reis

Texas A&M University
gdr@cse.tamu.edu

Abstract. We present the design and implementation of a system for axiomatic programming, and its application to mathematical software construction. Key novelties include a direct support for user-defined axioms establishing local equalities between types, and overload resolution based on equational theories and user-defined local axioms. We illustrate uses of axioms, and their organization into *concepts*, in structured generic programming as practiced in computational mathematical systems.

1 Introduction

We want our programs to look as much as possible like the mathematical formulation of the algorithms they implement. If the current state of our mathematical software is of any indication, an abstraction gap between a published algorithm and its realization as computer program grows to the point where we have trouble convincing ourselves that a program really does what it is supposed to. Of course, this situation is not unique to computer algebra. It is a general problem in the software industry. However, computer algebra deals with entities with rich mathematical structures. Therefore, it is a natural place to try to understand and to attempt to solve the abstraction gap problem.

The weak programming language support for direct expression of mathematical algorithms as computer codes has baffling consequences, and may at times prove to render some of our computer algebra systems quite embarrassing tools for education. For example, in the AXIOM computer algebra system [13] (and its derivatives), a domain that satisfies the `AbelianMonoid` category does not necessarily satisfy those of the `Monoid` category. Yet, we don't think a freshman student could easily get away with pretending that an Abelian monoid is not a monoid. The problem has been known for quite a long period of time. It has several root causes. The most fundamental one was the lack of programming language features to express concise mathematical properties of operations in ways that could be effectively used by compilers. In the case of AXIOM, the notion of *axiom* was considered in the early 1980s but never implemented [2]. With the lack of linguistic features to directly express that some operations obey certain laws, programmers have developed elaborate schemes, often culminating with conferring to names — *i.e.* syntax — far more importance than semantics, at the very expense of mathematical correctness. As a result, we have seen proposals (and software) that use `AdditiveMonoid` to represent the mathematical idea of Abelian monoid, where the operation is decided once for all to be named `+`, and the name `MultiplicativeMonoid` to designate a monoid that is not necessarily known to be Abelian, where its operation

is decided (again) once for all to be named $*$. As to be expected, one runs into serious problems when the same datatype carries simultaneously several instances of the same abstract structure. For example, the set of integers is obviously an Abelian monoid with respect to both usual $+$ and $*$, and we do want our software to reflect the fact that the operation $*$ is commutative. Furthermore, those are not the only Abelian monoid operations on integers; gcd is another example; so is the operation φ defined by

$$\varphi(x, y) = x + y + xy.$$

In short, the workaround that consists of conferring more importance to syntax than semantics is not just conceptually incorrect: it does not scale.

The main contribution of this paper is the design and implementation of a system (named Liz) that directly supports axioms, and structured generic programming as practiced in computer algebra. The system is an extension of a subset of the popular programming language C++ [21][2]. The long term goal is to bring structured generic programming to mainstream and accessible to “ordinary programmers”. This can be achieved if we have enough automation. The extensions are “*concepts*” [7][10] and “*axioms*” [9]. A key design concern has been to reduce the amount of “boilerplate” that programmers have to write in order for the compiler to “understand” that their programs have structures. The core of the type system is based on equational reasoning and axioms introduced by users. A central problem is: How can the compiler use user-defined constraint to resolve calls to overloaded functions? Overall, the Liz system offers a remarkable application of symbolic mathematics and deduction techniques [27] to the field of programming language design and implementation. Liz’s type checker combines pattern matching and automated logical deduction.

The remaining of this paper is structured as follows. We introduce axiomatic programming and the key language features behind Liz in (Section 2.) The general architecture, and the problems of Type checking, code generation, and constraint satisfaction are discussed in Section 3. Finally we discuss related work and we conclude in Section 4.

2 Axiomatic Programming

We define *axiomatic programming* as the practice of structured generic programming [4][3][15][16][5][19] that expresses and uses axioms or mathematical properties directly in code. The idea has been recently illustrated by Stepanov and McJones [19]. They showcased programming as a mathematical activity, a wonderful journey in the land of simplicity and generality. Their approach and exposition makes essential use of *properties* and *concepts*. In spite of this, all of their codes is compilable as almost C++03 [12] program fragments. There are two reasons for this. First, although the exposition is centered around concepts, the book never actually presents any single concept in concrete code — concepts were defined as abstract mathematical entities. From the C++ compiler point of view, concepts — the propellers — are effectively equivalent to comments, *i.e.* white spaces. Second, the actual computer codes use a few simple macros. In particular, the `requires` “keyword” is actually a C99 variadic macro defined [19, Appendix B.2] that ignores its arguments. Consequently, one of the benefits of concepts

— turning informal descriptions into codes that can be verified and used for type checking template definitions and uses — is not realized. The next subsections introduce the programming system Liz designed to support axiomatic programming.

2.1 Axioms and Concepts

In Liz, an axiom is a property that states what an algorithm may assume of values, operations, etc. but also what a user should and should not assume about these entities. Those properties are logical expressions about semantics we assume to hold for a proper execution of an algorithm. As such, what we call an axiom in Liz corresponds to the usual mathematical practice of “hypothesis forming” in developing a theorem. For example the notion of associativity is expressed in Liz as

```
template<BinaryOperation Op>
  axiom associative(Op op) {
    forall(Domain(Op) a, Domain(Op) b, Domain(Op) c)
      op(op(a, b), c) == op(a, op(a, b));
  }
```

That is, the axiom `associative` can be applied to a specific binary operation — not all binary operations are associative. The `template` header is used here to introduce the type `Op` of the parameter `op`. It also indicates that when used as in

```
associative(gcd);
```

Liz should magically figure out the value for `Op` — assuming `gcd` is not overloaded. The “type” `BinaryOperation` of the template parameter `Op` is a concept.

A concept [719] is a collection of properties (existence of operations, values, semantics, etc.) about types, operations, and values. For example the notion of a magma operation is rendered in Liz as the concept `BinaryOperation`. In general, concepts are organized in hierarchies. A magma operation is a binary homogeneous function, *i.e.* a function whose argument types are identical. In Liz, that concept is expressed as

```
concept HomogenousFunction(Function F) {
  Arity(F) > 0;
  forall(int i, int j) i < Arity(F) and j < Arity(F) =>
    InputType(F, i) == InputType(F, j);
}
```

which says that a function type `F` satisfies the requirements of `HomogenousFunction` if and only if, it is of positive arity (*i.e.* functions of that type take at least one argument), and all parameter types are identical. The function `InputType` is a builtin binary function that returns the argument type of a function. It is an example of what we call *type function*, *i.e.* type producing functions. Programmers can define their own type functions, as we will shortly see. Notice that we use a logical formula to state that property at a sufficiently abstract level. The homogeneity property implies that it makes sense to define the following function

```
typename Domain(HomogenousFunction F) {
  return InputType(F, 0);
}
```

Here, `Domain` is a function that accepts any homogenous function type argument and returns the common argument type. Notice the use of `typename` as return type. Indeed, `typename` is a concept that designates the collection of all types in `Liz`. Type functions can be evaluated at both compile and run time.

We refine further the notion of homogenous function to that of operation, where the domain is the same as the target

```
concept Operation(HomogenousFunction Op) {
    Codomain(Op) == Domain(Op);
}
```

Here, we use a builtin type function, `Codomain`, to specify that constraint on the return type. Again, so far the expressions of the concepts are very close to their abstract mathematical statements. Finally, we define a magma operation as an operation of arity 2.

```
concept BinaryOperation(Operation Op) {
    Arity(Op) == 2;
}
```

As an illustration of use, here is how in `Liz` one defines a function that applies a binary operation to the same argument

```
template<BinaryOperation Op>
Domain(Op) square(const Domain(Op)& x, Op op) {
    return op(x, x);
}
```

As defined, the `square` function can be applied to any binary `Liz` function like in

```
int mult(int x, int y { return x * y; }
int i = square(4, mult);
```

but also to any binary function object like in

```
int j = square(3, plus<int>());
```

where `plus<int>` is the equivalent of C++ standard function object of the same name. This versatility is possible precisely because the specification of `BinaryOperation` is abstract enough to allow several concrete representations of operations to be used. Yet, it is also precise enough to allow for complete type checking at the definition site — unlike the case of traditional C++ templates.

2.2 Associated Types and Values

Algebraic structures are usually determined by fundamental operations and derived or *associated values and types*. For instance, a monoid structure is uniquely defined by a monoid operation. In turn the neutral element of a monoid structure is uniquely associated with the monoid operation, *i.e.* the neutral value of a monoid is a function of the monoid operation. The notion of neutral value can be generally defined for any binary operation:

```
template<BinaryOperation Op>
  axiom has_neutral(Op op, Domain(Op) e) {
    forall(Domain(Op) x)
      op(x, e) == x and op(e, x) == x;
  }
```

From there, the notion of monoid operation comes as:

```
concept MonoidOperation(BinaryOperation Op) {
  associative(op);
  exist(Domain(op) e) {
    has_neutral(op, e);
    using neutral_value = e; // name it from now on
  }
}
```

Here, we express a monoid operation as a binary operation that is associative and that happens to admit a left- and right-neutral value. Furthermore, the `MonoidOperation` definition introduces the name `neutral_value` as an alias for that neutral value. It is clear that alias *depends* on the operation `op`, as indicated by existential quantification. Associated values and types can be referred to outside concepts definitions using a functional notation. The Liz compiler internally uses Skolemization to process associated entities. For example, after the statement

```
assume has_neutral(gcd, 0);
```

the Liz compiler can determine that the operation `gcd` satisfies the `MonoidOperation` concept and that `neutral_value(gcd)` is `0`.

3 Implementation

Given the abstract level of specification of operations in Liz, it is natural to expect the implementation to depart from conventional type checking. Below, we describe the challenges we face and solutions currently implemented in Liz. Despite being based on C++, Liz's template system behaves quite differently from standard C++ templates. Indeed, in C++ the following function template

```
template<typename T>
  T twice(T x) {
    return x + x;
  }
```

is perfectly fine, even though the compiler cannot determine all possible `operator+` that could be used. That decision is deferred until the function template `twice` is instantiated.

In Liz, that template definition will not type check. The elaborator objects with:

```
no match for operation 'operator+' with argument
  type list (T, T)
candidates are
  operator+: (int, int) -> int
  operator+: (double, double) -> double
```

This difference is deliberate. One of our goal is to understand what it takes to support axiomatic programming with C++-like template-like system with early checking by default.

3.1 Structure of the Liz System

The Liz system is designed with careful phase distinction considerations in mind. However, it is currently implemented as an interactive read-eval-print system. An input program fragment is decomposed into a token stream by a lexer. A parser arranges the token stream into a sequence of AST objects following the Liz grammar. Then, an elaborator takes the AST object sequence and type checks it, generating an alternative representation of the input program in a much simpler expression-based intermediate language. The intermediate expression language is designed in such a way that the evaluator does not require type information, *i.e.* the elaboration phase implements a type-erasure semantics. This design choice reflects our desire to ultimately reflect core C++ semantics and efficiency. Liz deliberately blurs the syntactic distinction between “type expression” and “ordinary expression”. This stems from the fact that Liz supports *type function*, e.g. functions (such as `Domain` from Section 2.1) which when applied to arguments produce types. Consequently, while the elaborator can most of the time determine through type checking that an expression designates a type, it occasionally needs some form of evaluation to reduce type function calls. Finally, the main feature of Liz — *axioms* — requires compile-time expression evaluation (involving function calls) to determine whether a combination of types and values satisfy certain logical formulæ. Compile-time evaluation of calls to user-defined functions with constant expressions is now part of standard C++ [8].

3.2 Type Checking

The main features of the Liz system are axioms and concepts. And that is the focus of this section.

Intermediate Language. The elaborator type checks and translate the input source program into an internal intermediate language. We briefly describe that intermediate language here for the benefit of the following subsections. First, it should be noted that the intermediate language is an expression-based language. This means that there is no arbitrary syntactic distinction between statements, expressions, or definitions. Every expression produces a value of one sort or the other.

Second, the intermediate language does not involve any notion of overloading. However, since Liz programs can overload functions, we must represent overloaded symbols in some way. We do this by pairing every symbol with its type (as computed by the elaborator).

Third, we present the syntax of intermediate language as s-expressions to ease visualization. Note, however, that Liz is itself implemented in ISO C++. The building blocks are:

- (`@formal p l n`) represents a formal parameter at position p , nesting level l , and named n
- (`@symbol n t`) represents a reference to symbol named n , and with declared type t
- (`@read a`) represents a read operation from a location designated by a
- (`@write a v`) represents a write operation to location a with value v
- (`@unary f x`) represents the call of a builtin unary operation f with argument x
- (`@binary f x y`) represents the call of a builtin binary operation f with arguments x and y
- (`and x y`) represents a conjunction logical formula
- (`or x y`) represents a disjunction logical formula
- (`=> x y`) represents an implication logical formula
- (`@call f x0 ... xn-1`) represents the call of a user defined operation f with arguments x_0, \dots, x_{n-1}
- (`@if p x y`) represents an if-statement with condition p , and branches x and y
- (`@while p x`) represents a do-statement with condition p , and body x
- (`@return v`) represents a return-statement with value v
- (`@block x0, ... xn-1`) represents a block composed of the statement sequence x_0, \dots, x_{n-1}
- (`@bind n t`) represents a symbol n with type t in the current frame.
- (`@forall (@parameters p0 ... pn-1) x`) represents a universally quantified logical formula x , and that binds the parameters p_0, \dots, p_{n-1}

In addition to these forms, there are representations for basic type, basic constants. Unary builtin and binary builtin operations are represented as

- (`@builtin n t`) where n is the name of the operation and t is its (function) type

Finally, because the implementation of the intermediate language is strongly typed, we use the form

- (`@type_expr e`) for expressions that represent types; this is typically the case when a type function is applied to archetypes (Section [3.2](#))

Elaborating Axioms. Axioms are first-order predicate formulae. They can make references to any user-defined function. Before type checking and code generation, an axiom is first put in prenex form. The body is then elaborated as a Boolean expression, where names bound by the quantifiers prefix are treated as if they were formal (function) parameters with the declared types. For example, the following axiom taken from the definition of `HomogenousFunction`

```
forall(int i, int j) i < Arity(F) and j < Arity(F) =>
  InputType(F, i) == InputType(F, j);
```

is elaborated as

```
(@forall (@parameters (@formal 0 1 i) (@formal 1 1 j))
  => (and (@binary
    (@builtin operator< (int, int) -> bool)
```

```

      (@formal 0 1 i)
      (@unary (@builtin Arity (Function) -> int)
        (@type_expr (@formal 0 0 F))))
    (@binary
      (@builtin operator< (int, int) -> bool)
      (@formal 1 1 j)
      (@unary (@builtin Arity (Function) -> int)
        (@type_expr (@formal 0 0 F))))))
  (@binary
    (@builtin operator== (typename, typename) -> bool)
    (@type_expr
      (@binary
        (@builtin InputType (Function, int) -> typename)
        (@type_expr (@formal 0 0 F))
        (@formal 0 1 i)))
    (@type_expr
      (@binary
        (@builtin InputType (Function, int) -> typename)
        (@type_expr (@formal 0 0 F))
        (@formal 1 1 j))))))

```

Notice that all overloaded operators have been resolved. It is this elaboration that is stored for future uses, in particular in deciding constraints satisfaction (see Section [3.2](#)).

Concept Elaboration. Recall that a concept is a collection of syntactic, semantics, and complexity requirements on a collection of operations, types, and values. In Liz, concepts are expressed as a sequence of axioms, refinements, and operation signature specifications. The result of elaborating a concept is a 5-tuple:

1. a concept name
2. a sequence of elaboration of parameters explicitly bound in the concept definition — we refer to these as explicit concept parameters
3. a sequence of elaborations of formulæ mentioned in the concept definition
4. a sequence of elaborations of refined concepts
5. a sequence of elaborations of signatures explicitly mentioned in the concept definition — we refer to these operations as implicit concept parameters.

An explicit concept parameter is elaborated just like any other kind of parameter. If the type mentioned in the parameter declaration is a (unary) concept, then the elaboration goes through an additional step called *dressing* to create an archetype as will be explained in Section [3.2](#).

Refined concepts are elaborated as predicates, with the understanding that they are properties that a function template definition can assume and use, while they acts as pre-conditions at a function template call site. These refined concepts are also used during dressing of archetypes.

An operation signature explicitly mentioned in a concept definition is to be thought of as a requirement, a proof obligation to be fulfilled at the point of the concept use (through function template call.) Consequently, they are handled as parameters — except that their values are implicitly deduced during constraint satisfaction.

The collection of explicit concept parameters and implicit concept parameters form the domain of the substitution that results from a successful constraint satisfaction checking. That substitution is then used to expand or instantiate the elaboration of the selected function template (see Section 3.3)

Archetypes and Dressing. Conventional type checking assumes that types are values that are known at compile-time. To deal with type variables (used by generic functions), one synthesizes an arbitrary value for the type variable, and type checking proceeds as usual. That arbitrary value is what we call *archetype* of a type parameter. It symbolizes any value that parameter may take on when the generic function is instantiated. At the basic level, the archetype does not have any properties, except that it is a type value. To be useful in generic algorithms, the archetype T needs to carry information useful for type-checking purposes. The process of endowing an archetype with additional assumptions is what we call *dressing*.

Dressing of an archetype T with a concept type \mathcal{C} is given by the following algorithm:

1. for each logical formula f in \mathcal{C} , simplify the instantiation of f , based on the existing property set of T . If the result is not vacuously true, add it to the property set of T .
2. for each refined concept \mathcal{C}' in \mathcal{C} , dress the archetype T with \mathcal{C}' .

Note that in abstract, the order in which properties are added to the property set of an archetype does not matter. However, from a practical point of view, we do want to keep property sets as small as possible for reasons that will become obvious by the end of the type equivalence subsection. We observe that in a concept hierarchy, a refining concept usually adds more information that restricts the collection of satisfying types. In particular, formulæ from refined concepts may get simpler with the addition of new constraints. We can see this with our `HomogenousFunction` example. With that concept, all we know is that the arity of any function type F that satisfies `HomogenousFunction` must be a positive integer. The definition concept of `BinaryOperation` gives a definite value to the arity. Consequently, the dressing of an archetype `Op` of `BinaryOperation` produces the following trace of its property set:

1. Start with $P_0 = \{\text{Arity}(\text{Op}) == 2\}$, which is the sole formula in `HomogenousFunction`
2. Dress `Op` with `Operation`

- (a) simplify the formula

$$\text{Codomain}(\text{Op}) == \text{Domain}(\text{Op})$$

with P_0 . This involves reducing each side of the equality operator in irreducible form. In particular the type function call `Domain(Op)` is reduced to `InputType(Op, 0)`. There is no other formula in P_0 that would reduce the formula.

- (b) Add `Codomain(Op) == InputType(Op, 0)` to P_0 to obtain the new property set

$$P_1 = \left\{ \begin{array}{l} \text{Arity}(\text{Op}) == 2, \\ \text{Codomain}(\text{Op}) == \text{InputType}(\text{Op}, 0) \end{array} \right\}$$

3. Dress Op with `HomogenousFunction`

(a) simplify

```
forall(int i, int j)
  i < Arity(Op) and j < Arity(Op) =>
    InputType(Op,i) == InputType(Op,j)
```

with the property set P_1 to obtain

```
forall(int i, int j)
  i < 2 and j < 2 =>
    InputType(Op,i) == InputType(Op,j)
```

Note that although we show the input-source form above, the simplification is really done on the *elaboration* of the formula. The simplification is implemented a typefull term rewrite engine.

(b) Add the resulting formula to P_1 to obtain

$$P_2 = \left\{ \begin{array}{l} \text{Arity(Op)} == 2, \\ \text{Codomain(Op)} == \text{InputType(Op,0)}, \\ \text{forall(int i, int j)} \\ \quad \text{i} < 2 \text{ and } \text{j} < 2 \Rightarrow \\ \quad \text{InputType(Op,i)} == \text{InputType(Op,j)} \end{array} \right\}$$

(c) simplify the formula $\text{Arity}(2) > 0$ with respect to the property set P_2 to obtain $2 > 0$, which is vacuously true. So the final property set of the archetype Op is P_2 .4. Dress Op with the builtin concept `Function`, which does not actually add any formula.

During the dressing procedure we interpret an equality formula $a == b$ as defining the expression a in terms of the expression b , even though the equality operator is in fact commutative. We could avoid this restriction by using unification algorithms that cope with commutativity. That is an improvement we consider as future work.

Type Equivalence. During type checking we need to determine when an expression of type T is acceptable in a context where a value of type S is expected. Conventional type checking solves this essentially as an equality problem in the free algebra generated by base types and type constructors.

That approach is inadequate for axiomatic programming. In fact, the defining trait of this style of programming is precisely that programmers can state in very abstract, if somewhat stylized fashion, relations between types. Consequently determining the equivalence of two type expressions amounts to determining identities in an equational theory. The set of equations to consider varies from one context to the next, depending on the set of assumptions in scope. Consider the square function example

```
template<BinaryOperation Op>
  Domain(Op) square(const Domain(Op)& x, Op op) {
    return op(x, x);
  }
```

Here, to check that the call `op(x, x)` is well-formed, we need to check that `x` can be used as both the first and second argument to the operation `op`. First, we discuss the type of `op`, then we move on the matching of arguments.

When the type checker sees the call `op(x, x)`, it first tries to determine that `op` is an expression that can be used in an operator position. The answer is yes, because the archetype `Op` satisfies `Function`. Next, it needs to determine its arity. This information is found by pattern matching the expression `Arity(Op) == n` against the formulæ in the property set of `Op`. Which gives the value 2. The elaborator then synthesizes the type

$$(\text{InputType}(\text{Op}, \emptyset), \text{InputType}(\text{Op}, 1)) \rightarrow \text{InputType}(\text{Op}, \emptyset)$$

The return type is the result of rewriting `Codomain(Op)` with respect to the property set P_2 . With this type for `op` the elaborator proceeds to check the arguments.

The type of the parameter `x` is `const Domain(Op)&`. This type expression involves a type function, `Domain`, which is defined only on types that satisfy `HomogenousFunction`. Since the archetype `Op` satisfies `BinaryOperation`, it also satisfies `HomogenousFunction`. Hence, evaluation of `Domain(Op)` is legitimate, and yields `InputType(Op, \emptyset)`. So the type of `x` is `const InputType(Op, \emptyset)&`.

Next, we need to determine if the use of `x` as first argument to `op` is well-formed. In general, the use of a reference in a non reference context implies a read operation. The read operation yields an expression of type `const InputType(Op, \emptyset)`. Finally, we observe that using a *value* of type `const T` in a context where a value of type `T` is expected is OK — in another term, it is fine to lose toplevel `const`-qualification on values.

We now need to determine whether the use of `x` as second argument of `op` is well-formed. We follow the same procedure as in the previous paragraph. Which leads to determining the equivalence `InputType(Op, \emptyset)` and `InputType(Op, 1)`. At this point, we consider again the property set P_2 . We obtain the equality we were looking for by considering the formula

$$\begin{aligned} &\text{forall}(\text{int } i, \text{int } j) \\ &\quad i < 2 \text{ and } j < 2 \Rightarrow \\ &\quad \text{InputType}(\text{Op}, i) == \text{InputType}(\text{Op}, j) \end{aligned}$$

and using standard deduction system based on sequent calculus. This deduction engine is part of the elaborator.

As can be seen from the example just discussed, every single type equivalence problem implies a search of a set of databases formed by property sets of relevant archetypes, intertwined with possible logical formulæ satisfiability. For this reason, it is beneficial to keep property sets sizes as small as possible.

Concept Satisfaction. The problem of concept satisfaction is a fruitful source of debates. Essentially, there are two schools of thoughts.

On the one hand, there is explicit conformance, that is the elaborator should consider that a type satisfies a concept only if there is an explicit statement to that effect — not just because some function declarations are in scope and predicates are satisfied. This is the approach implemented by the AXIOM system (and its variants including Aldor),

and the Haskell programming language. It certainly is the favorite approach in certain type theory circles. A problem with this approach, in our opinion, is that it does not scale well in practice. One of the key aspects of C++ templates, that contributed to the success of the Standard Template Library [18], is the implicit matching of interfaces — or “duck typing” as it is called. Also, we believe that this approach has a deep justification rooted in the lack of language features to differentiate operations based on mathematical properties (or lack thereof.)

On the other hand, we have the notion of implicit conformance: a type satisfies a concept if it meets all its predicates, and all signatures have matching concrete function definitions. This is the approach we take for the Liz system. To determine that a type τ satisfies a concept \mathcal{C} , we use the following algorithm:

1. Instantiate \mathcal{C} by substituting τ for its parameter. Simplify all logical formulæ. If any refined concept of \mathcal{C} is not satisfied, then satisfaction of \mathcal{C} fails.
2. For each signature specification in \mathcal{C} , try to find a matching declaration in scope. If the matching fails, or has more than one solution, satisfaction of \mathcal{C} fails. This step finds values for the implicit parameters.
3. Substitute implicit parameters in logical formulæ in \mathcal{C} , if any of them is not satisfiable then satisfaction of \mathcal{C} fails.

3.3 Code Generation

There are several code generation techniques for handling generic functions. We briefly mention two, which are used in mainstream programming languages that support generic programming.

A common technique is to associate with each generic function a vector of used operations (or dictionary) that maps abstract operations to their concrete implementations. When a generic function is called, a dictionary argument is constructed and passed as an implicit additional argument. This technique is used in the implementation of the AXIOM system [13], in the implementation of Aldor, and is the conventional implementation [111] of the Haskell programming language [17] for its type classes features [26]. This technique is very attractive in the sense that code generated for a program that uses a generic function contains only one definition or instance of that function, no matter how many times it is statically used. The technique also supports separate compilation. However, in practice it does bring a non-negligible abstraction penalty. This is because every (abstract) operation used in a generic function (no matter how cheap its concrete realization is) is looked up at runtime through the dictionary argument. It is obvious that the cost of this implementation technique is unacceptable for certain operations. There are a number of implementation tricks to reduce the dynamic lookup overhead. For example, if it is known that an abstract operation designates the same concrete realization during the execution of a generic function call, the result of the first lookup can be cached and reused during the lifetime of that particular call instance. But, it is equally clear that for a modular algorithm that operates directly on machine integers, the cost of adding (or multiplying) two machine integers becomes prohibitive if the integer operation is not directly inlined, resulting in simple machine code [7]. The Aldor programming language uses the notion of *domain inlining* to help programmers

work around this efficiency issue. The idea is that a programmer, after careful analysis, would single out domains that are tightly coupled with generic function (mostly for efficiency reasons). The compiler will then resolve statically all calls to operations implemented by those domains, thereby bypassing the runtime dictionary lookup overhead. The technique is effective. However, separate compilation is lost because the resulting function now depends on implementation details of the domain. Moreover, we believe it requires a fair amount of foresight from programmers, especially library writers; and it is not clear how that approach scales when independently developed components (written by several independent groups of people) are composed.

The second code generation technique for generic function expands every generic function reference (but unique in its parameter types) into a new and distinct copy of the original generic function, where abstract parameters are replaced by their concrete values. The result is then subject to normal conventional code generation technologies. This technique is popular with C++ template implementations and some implementations of Ada generics [25]. Codes generated for generic functions by this approach are near optimal. However, it should be observed this technique may lead to code bloat in case of undisciplined programs that are not properly organized to take advantage of structural and semantics commonalities. In the case of C++ template, this technique can actually lead to code bloat removal, as surprising as it may come contrary to the conventional wisdom. The reason is very simple: a template function is instantiated if and only if it is used. That is, the C++ language type system contains explicit provision for “dead code” removal for generic function instances.

The second code generation technique just discussed is the approach we use in the Liz system. However it differs from conventional C++ template compilation in the sense that we expand the result of elaboration. That is, a function template definition is fully type checked, with corresponding generated code. It is the result of that elaboration that is expanded when the function is called. The concept system as currently designed and implemented ensures that no type error will occur at code expansion time. This is to be contrasted with what happens with current C++ templates. We achieve this result by rejecting some popular implicit conversions (mostly between basic types such as `bool`, `int`, `double`), which insures that expression types are preserved under substitution. We fully acknowledge that this restriction rules out many real world C++ programs, but the purpose of the Liz system is not to emulate anarchic type conversions [20].

4 Related Work and Conclusion

There is a large body of work in the general area of advanced language features for generic programming but, to our knowledge, none in major use today aims at direct support for axiomatic programming. Within the computer algebra community, Jenks and Trager [14] explained the design and implementation of the Scratchpad system, which later became AXIOM. The Aldor programming language (the better version of the AXIOM library extension language) aims at a more categorial view of programs. However, it does not have support for axiomatic constructions. With the C++ community, the most relevant work is the collaborative effort [7,10] to introduce concepts into C++0x. While axioms were identified [6] early as key aspects of concepts, they

are formally proposed only very late in the process [9]. At that point, the C++ concepts proposal was already exhibiting worrisome complexities that would prompt its eventual removal [22,23] from the C++0x draft. Our opinion is that a good part of the complexity came from the fact that the proposal did not provide good enough support for more abstract definitions of algorithms. For a language as complex as C++ that has been in industrial use for nearly 3 decades, a successful proposal to support axiomatic programming must abstract over details, as opposed to aiming at reflecting them. Andrew Sutton and Bjarne Stroustrup recently begun investigation of semantics-oriented libraries for C++ [24]. The Liz system was designed to support more abstract generic algorithm definitions. It needs a terrain for experimentations, and computer algebra is a natural testbed — for it is hard to quibble with maths (semantics). Its implementation is a remarkable application of computer algebra and symbolic mathematics techniques.

Acknowledgment. I would like to thank Bjarne Stroustrup, Erik Katzen, Jasson Casey for comments on earlier drafts of this paper. Carla Villoria contributed to an early implementation of the Liz parser. This work was partially supported by NSF grants CCF-1035058 and CCF-1150055.

References

1. Augustsson, L.: Implementing Haskell Overloading. In: Functional Programming Languages and Computer Architecture, pp. 65–73 (1993)
2. Davenport, J.H.: Private communication (May 2009)
3. Davenport, J.H., Gianni, P., Trager, B.M.: Scratchpad’s View of Algebra II: A Categorical View of Factorization. In: ISSAC 1991: Proceedings of the 1991 International Symposium on Symbolic and Algebraic Computation, pp. 32–38. ACM Press, New York (1991)
4. Davenport, J.H., Trager, B.M.: Scratchpad’s View of Algebra I: Basic Commutative Algebra. In: Miola, A. (ed.) DISCO 1990. LNCS, vol. 429, pp. 40–54. Springer, Heidelberg (1990)
5. Dehnert, J.C., Stepanov, A.: Fundamentals of Generic Programming. In: Jazayeri, M., Musser, D.R., Loos, R.G.K. (eds.) Dagstuhl Seminar 1998. LNCS, vol. 1766, pp. 1–11. Springer, Heidelberg (2000)
6. Reis, G.D.: Generic Programming in C++: The next level. In: The Association of C and C++ Users Spring Conference (April 2002)
7. Reis, G.D., Stroustrup, B.: Specifying C++ Concepts. In: Conference Record of POPL 2006: The 33th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, pp. 295–308 (2006)
8. Reis, G.D., Stroustrup, B.: General Constant Expressions for System Programming Languages. In: Proceedings of the 25th Symposium on Applied Computing, Sierre, Switzerland, pp. 2133–2138. ACM Press (March 2010)
9. Reis, G.D., Stroustrup, B., Meredith, A.: Axioms: Semantics Aspects of C++ Concepts. Technical Report N2887=09-0077, ISO/IEC SC22/JTC1/WG21 (June 2009), <http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2009/n2887.pdf>
10. Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Reis, G.D., Lumsdaine, A.: Concepts: Linguistic Support for Generic Programming in C++. In: OOPSLA 2006: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, and Applications, pp. 291–310. ACM Press, New York (2006)
11. Hall, C.V., Hammond, K., Jones, S.P., Wadler, P.L.: Type classes in Haskell. ACM Transactions on Programming Languages and Systems 18(2), 109–138 (1996)

12. International Organization for Standards. International Standard ISO/IEC 14882. Programming Languages — C++, 2nd edn. (2003)
13. Jenks, R.D., Sutor, R.S.: AXIOM: The Scientific Computation System. Springer (1992)
14. Jenks, R.D., Trager, B.M.: A Language for Computational Algebra. SIGPLAN Not. 16(11), 22–29 (1981)
15. Musser, D.A., Stepanov, A.A.: Generic Programming. In: Gianni, P. (ed.) ISSAC 1988. LNCS, vol. 358, pp. 13–25. Springer, Heidelberg (1989)
16. Musser, D.R., Stepanov, A.: Algorithm-oriented Generic Libraries. Software-Practice and Experience 24(7), 623–642 (1994)
17. Jones, S.P.: Haskell 98 Language and Libraries, The Revised Report. Cambridge University Press (2003)
18. Stepanov, A., Lee, M.: The Standard Template Library. Technical Report N0482=94-0095, ISO/IEC SC22/JTC1/WG21 (May 1994)
19. Stepanov, A., McJones, P.: Elements of Programming. Addison-Wesley (2009)
20. Stroustrup, B.: The Design and Evolution of C++. Addison-Wesley (1994)
21. Stroustrup, B.: The C++ Programming Language, special edn. Addison-Wesley (2000)
22. Stroustrup, B.: Simplifying the use of concepts. Technical Report N2906, ISO/IEC SC22/JTC1/WG21 (June 2009)
23. Stroustrup, B.: The C++0x "Remove Concepts" Decision. Dr. Dobb's Journal (2009), <http://www.ddj.com/cpp/218600111?pgno=1>, Republished with permission in Overload Journal 92 (August 2009)
24. Sutton, A., Stroustrup, B.: Design of Concept Libraries for C++. In: Sloane, A., Abmann, U. (eds.) SLE 2011. LNCS, vol. 6940, pp. 97–118. Springer, Heidelberg (2012)
25. Tucker Taft, S., Duff, R.A., Brukardt, R.L., Plödereder, E. (eds.): Consolidated Ada Reference Manual. LNCS, vol. 2219. Springer, Heidelberg (2001)
26. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Austin, Texas, USA, pp. 60–76 (1989)
27. Watt, S.: What Happened to Languages for Symbolic Mathematical Computation? In: Proceedings of Programming Languages for Mechanized Mathematics (PLMMS), Hagenberg, Austria, June 29-30, pp. 81–90. RISC-Linz (2007)
28. Weibel, T., Gonnet, G.H.: An Assume Facility for CAS, with a Sample Implementation for Maple. In: Fitch, J. (ed.) DISCO 1992. LNCS, vol. 721, pp. 95–103. Springer, Heidelberg (1993)

Reasoning on Schemata of Formulæ^{*}

Mnacho Echenim and Nicolas Peltier

University of Grenoble (LIG, Grenoble INP/CNRS)

{Mnacho.Echenim,Nicolas.Peltier}@imag.fr

Abstract. A logic is presented for reasoning on iterated sequences of formulæ over some given base language. The considered sequences, or *schemata*, are defined inductively, on some algebraic structure (for instance the natural numbers, the lists, the trees etc.). A proof procedure is proposed to relate the satisfiability problem for schemata to that of finite disjunctions of base formulæ. It is shown that this procedure is sound, complete and terminating, hence the basic computational properties of the base language can be carried over to schemata.

1 Introduction

We introduce a logic for reasoning on iterated schemata of formulæ. The schemata we consider are infinite sequences of formulæ over a given *base language*, and these sequences are defined by induction on some algebraic structure (e.g. the natural numbers). As an example, consider the following sequence of propositional formulæ ϕ_n , parameterized by a natural number n :

$$\phi_0 \rightarrow \top \quad \phi_{n+1} \rightarrow \phi_n \wedge (p(n) \Leftrightarrow p(n+1)).$$

It is clear that the formula $\phi_n \wedge p(0) \wedge \neg p(n)$ is unsatisfiable, for every $n \in \mathbb{N}$. This can be easily checked by any SAT-solver, for every *fixed* value of n . Here the base language is propositional logic and the sequence is defined over the natural numbers. However, proving that it is unsatisfiable *for every* $n \in \mathbb{N}$ is a much harder task which obviously requires the use of mathematical induction. Similarly, consider the sequence:

$$\psi_{nil} \rightarrow \top \quad \psi_{cons(x,y)} \rightarrow \psi_y \wedge (\exists u p(y, u) \Leftrightarrow (\exists v p(cons(x, y), v)))$$

Then $\psi_l \wedge p(nil, a) \wedge \forall u \neg p(l, u)$ is unsatisfiable, for every (finite) list l . Here the base language is first-order logic and the sequence is defined over the set of lists. Such inductively defined sequences are ubiquitous in mathematics and computer science. They are often introduced to analyze the complexity of proof procedures. From a more practical point of view, schemata of propositional formulæ are used to model properties of circuits parameterized by natural numbers, which can represent, e.g., the number of bits, number of layers etc. (see for instance [15], where

^{*} This work has been partly funded by the project ASAP of the French *Agence Nationale de la Recherche* (ANR-09-BLAN-0407-01).

a language is introduced to denote inductively defined boolean functions which can be used to model such parameterized circuits). In mathematics, schemata of first-order formulæ can model inductive proofs, which can be seen as infinite (unbounded) sequences of first-order formulæ (see [5] for an example of the use of this technique in proof analysis).

We now provide a slightly more complex example. The following schema ψ_t encodes a multiplexer, inductively defined as follows. The base case is denoted by $Base(x)$, where x denotes an arbitrary signal. In this case, the output of the circuit is simply the output of x , denoted by $signal(x)$. The inductive case is denoted by $Ind(i, x, y)$, where i is a select input and x and y are two smaller instances of the multiplexer. Its output is either the output of x or that of y , depending on the value of i .

$$\begin{aligned} \psi_{Base(x)} &\rightarrow out(Base(x)) \Leftrightarrow signal(x) \\ \psi_{Ind(i,x,y)} &\rightarrow (\neg signal(i) \vee (out(Ind(x,y)) \Leftrightarrow out(x))) \\ &\quad \wedge (signal(i) \vee (out(Ind(x,y)) \Leftrightarrow out(y))) \\ &\quad \wedge \psi_x \wedge \psi_y \end{aligned}$$

Note that this kind of circuit cannot be encoded in the language of (regular) propositional schemata defined in [2,3], because the number of inputs is exponential in the depth of the circuit. Hence, the use of non-monadic function symbols is mandatory.

In this paper, we devise a proof procedure to check the satisfiability of these sequences. More precisely, we introduce a formal language for modeling sequences of formulæ defined over an arbitrary base language (encoded as first-order formulæ interpreted in some particular theory) and we show that the computational properties of the base logic carry over to these schemata: If the satisfiability problem is decidable (resp. semi-decidable) for the base language then it is also decidable (resp. semi-decidable) for the corresponding schemata. For instance, the satisfiability problem is decidable for schemata of propositional formulæ and semi-decidable for schemata of first-order formulæ. The basic principle of our proof procedure consists in relating the satisfiability of any iterated schemata of formulæ to that of a *finite* disjunction of base formulæ. The complexity of the satisfiability problem, however, is not preserved in general, since the number of formulæ in the disjunction may be exponential.

This work generalizes previous results [2,3] in two directions: first the base language is no longer restricted to propositional logic¹ and second the sequences are defined over arbitrary algebraic structures, and not only over the natural numbers. Abstracting from the base language leads to an obvious gain in applicability since our approach now applies to any logic, provided a proof procedure exists for testing the satisfiability of base formulæ. Besides, it has the advantage that the reasoning on schemata is now clearly separated from the reasoning on formulæ in the base language, which may be postponed. This should make our approach much more scalable, since any existing system could now be used as a

¹ A first extension to some decidable theories such as Presburger arithmetic was considered in [4].

“black box” to handle the basic part of the reasoning (whereas the two aspects were closely interleaved in our first approach, yielding additional computational costs). Both extensions significantly increase the scope of our approach.

The extension to arbitrary structures turns out to be the most difficult from a theoretical point of view, mainly because, as we shall see, the number of parameters can increase during the decomposition phase, yielding an increase of the number of related non-decomposable formulæ in each branch, which can in principle prevent termination. In contrast to what happens in the simpler case of propositional schemata [2], these formulæ *cannot* in general be deleted by the purity principle, since they are not independent from the other formulæ in the branch. To overcome this problem, we devise a specific instantiation strategy based on a careful analysis of the depth of terms represented by the parameters, and we define a new loop detection mechanism. This blocking rule is more general and more complex than the one in [2]. We show that it is general enough – together with the proposed instantiation strategy – to ensure termination. Termination is however much more difficult to prove than for propositional schemata defined over natural numbers.

The types of structures that can be handled are quite general: they are defined by sets of – possibly non-free – constructors on a sorted signature. The terms can possibly contain elements of a non-inductive sort. For instance, a list may be defined inductively on an *arbitrary* set of elements.

Related Work

There exist many logics and frameworks in which the previous schemata can be encoded, for instance higher-order logic [7], first-order μ -calculus [18], or logics with inductive definitions [1] that are widely used in proof assistants [19]. However, the satisfiability problem is not even semi-decidable for these logics (due to Gödel’s famous result). Very little published research seems to be focused on the identification of complete subclasses and iterated schemata definitely do not lie in these classes and cannot be reduced to them either. Our approach ensures that the basic computational properties of the base language (decidability or semi-decidability) are preserved, at the cost of additional restrictions on the syntax of the schemata under consideration. Furthermore, the modeling of schemata in higher-order languages, although possible from a theoretical point of view, is cumbersome and not very natural in practice.

There exist several approaches in inductive theorem proving, ranging from explicit induction approaches (see for instance [11] or [6]) used mainly by proof assistants to implicit induction schemes used in rewrite-based theorem provers [8,9], or even to inductionless induction [16,12], where inductive validity is reduced to a mere satisfiability check. Such approaches can in principle handle some of the formulæ we consider in the present work, provided the base language can be axiomatized. Existing approaches are usually only complete for refutation, in the sense that false conjectures can be disproved, but that inductive theorems cannot always be recognized (this is theoretically unavoidable). Once again, very few termination results exist for such provers and our language

does not fall in the scope of the known complete classes (see for instance [14]). In general, inductive theorem proving requires strong human guidance, especially for specifying the needed inductive lemmata. In contrast, our procedure is *purely automatic*. Of course, this comes at the expense of strongly reducing the form of the inductive axioms. Furthermore, although very restricted to ensure termination and/or completeness, our language allows for more general queries, possibly containing nested quantifiers, which are in general out of the scope of existing automated inductive theorem provers. Indeed, most existing approaches aim at establishing the inductive validity of universal queries w.r.t. a first-order axiomatization (usually a set of clauses). In contrast, our method can handle more general goals of the form $\forall \mathbf{x} \phi$, where \mathbf{x} is a vector of variables interpreted over the considered algebraic structure and ϕ is a formula containing arbitrary quantifiers in the base language.

Practical attempts to use existing inductive theorem provers (such as ACL [10]) to check the satisfiability of schemata such as those in the Introduction fail for every formula except the most trivial ones. We believe that this is not due to a lack of efficiency, but rather to the fact that additional inductive lemmata are required, which cannot be generated automatically by the systems. In some sense, our method (and especially the loop detection rule) can be viewed as an automatic way to generate such lemmata. Our method is also more modular: we make a clear distinction between the reasoning over the base logic and the one over inductive definitions. Inference rules are devised for the latter and an external prover is used to establish the validity of formulæ in the base language.

Since parameterized schemata can obviously be seen as monadic predicates, a seemingly natural idea would be to encode them in monadic second-order logic and use an automata-based approach (see, e.g., [17]) to solve the satisfiability problem. However, as we shall see in Section 3, the unfolding of the inductive definitions contained in a given formula may well increase the number of parameters occurring in it. Since these parameters may share subterms, the formulæ containing them are *not* independent hence they must be handled simultaneously, in the same branch. Thus a systematic decomposition into monadic atoms (in the style of automata-based approaches) is not feasible.

Due to space restrictions, the proofs are omitted and can be found in [13].

2 A Logic for Iterated Schemata

The schemata we consider in this paper are encoded as first-order formulæ, together with a set of rewrite rules specifying the interpretation of certain monadic predicate symbols. Our language is *not* a subclass of first-order logic: indeed, some sort symbols will be interpreted on an inductively defined domain (e.g. on the natural numbers). Furthermore, the formulæ can be interpreted modulo some particular theory, specified by a class of interpretations.

We first briefly review usual notions and notations. We consider first-order terms and formulæ defined on a sorted signature. Let \mathcal{S} be a set of *sort symbols*. Let Σ be a set of *function symbols*, together with a function *profile* mapping

every symbol in Σ to a unique non-empty sequence of elements of \mathcal{S} . We write $f : \mathbf{s}_1 \times \dots \times \mathbf{s}_n \rightarrow \mathbf{s}$ if $profile(f) = \mathbf{s}_1, \dots, \mathbf{s}_n, \mathbf{s}$ with $n > 0$, and $a : \mathbf{s}$ if $profile(a) = \mathbf{s}$ (in this case a is a *constant symbol*). A symbol is of sort \mathbf{s} and of arity n if its profile is of the form $\mathbf{s}_1, \dots, \mathbf{s}_n, \mathbf{s}$ (possibly with $n = 0$). The set of function symbols of sort \mathbf{s} is denoted by $\Sigma_{\mathbf{s}}$. Let $(\mathcal{V}_{\mathbf{s}})_{\mathbf{s} \in \mathcal{S}}$ be a family of pairwise disjoint sets of *variables of sort \mathbf{s}* , and $\mathcal{V} \stackrel{\text{def}}{=} \bigcup_{\mathbf{s} \in \mathcal{S}} \mathcal{V}_{\mathbf{s}}$. We denote by $T_{\mathbf{s}}$ the sets of *terms of sort \mathbf{s}* built as usual on Σ and \mathcal{V} . A term not containing any variable is *ground*.

Definition 1. Let \mathcal{I} be a subset of \mathcal{S} . The elements of \mathcal{I} are called the inductive sorts. An \mathcal{I} -term is a term of a sort $\mathbf{s} \in \mathcal{I}$.

Let $\mathcal{C} \subseteq \Sigma$ be a set of constructors, such that the sort of every symbol in \mathcal{C} is in $\bigcup_{\mathbf{s} \in \mathcal{I}} \Sigma_{\mathbf{s}}$ and such that every non-constant symbol of a sort in $\bigcup_{\mathbf{s} \in \mathcal{I}} \Sigma_{\mathbf{s}}$ is in \mathcal{C} . A parameter is a constant symbol of a sort occurring in the profile of a constructor (parameters are denoted by upper-case letters). A term containing only function symbols in \mathcal{C} and variables of sorts in $\mathcal{S} \setminus \mathcal{I}$ is a constructor term.

Constructors of a sort $\mathbf{s} \in \mathcal{I}$ are meant to define the domain of \mathbf{s} , see Definition 5. The constant symbols that are not constructors can be seen as existential variables denoting arbitrary elements of a sort in \mathcal{I} (notice however that \mathcal{C} possibly contains constant symbols). We assume that \mathcal{I} contains a sort symbol \mathbf{nat} , with two constructors $0 : \mathbf{nat}$ and $\text{succ} : \mathbf{nat} \rightarrow \mathbf{nat}$.

Example 1. Assume that we intend to reason on lists of elements of an arbitrary sort \mathbf{s} . Then \mathcal{S} contains the sort symbols \mathbf{s} and \mathbf{list} , where $\mathcal{I} = \{\mathbf{list}\}$. The constructors are $\text{nil} : \mathbf{list}$ and $\text{cons} : \mathbf{s} \times \mathbf{list} \rightarrow \mathbf{list}$. The set of parameters contains constant symbols of sorts \mathbf{s} or \mathbf{list} (denoting respectively elements and lists). If A_1, A_2 are parameters of sort \mathbf{s} , then $\text{cons}(A_1, \text{cons}(A_2, \text{nil}))$ is a term of sort \mathbf{list} .

Similarly, if one wants to reason on lists of natural numbers, then one should take $\mathcal{I} = \mathcal{S} = \{\mathbf{nat}, \mathbf{list}\}$. In this case, $\mathcal{C} = \{\text{nil} : \mathbf{list}, \text{cons} : \mathbf{nat} \times \mathbf{list} \rightarrow \mathbf{list}, 0 : \mathbf{nat}, \text{succ} : \mathbf{nat} \rightarrow \mathbf{nat}\}$.

Let $(\mathcal{D}_{\mathbf{s}})_{\mathbf{s} \in \mathcal{I}}$ be a family of disjoint sets of *defined symbols* of sort \mathbf{s} , disjoint from Σ , and $\mathcal{D} \stackrel{\text{def}}{=} \bigcup_{\mathbf{s} \in \mathcal{I}} \mathcal{D}_{\mathbf{s}}$. An *atom* is either an *equation* of the form $t \simeq s$, where t, s are terms of the same sort, or a *defined atom*, of the form d_t , where $d \in \mathcal{D}_{\mathbf{s}}$, for some $\mathbf{s} \in \mathcal{I}$, and $t \in T_{\mathbf{s}}$. The arguments of the symbols in \mathcal{D} are written as indices in order to distinguish them from predicate symbols that may occur in Σ (such predicate symbols may be encoded as functions of profile $\mathbf{s} \rightarrow \mathbf{bool}$). Formulæ are built as usual on this set of atoms using the connectives $\vee, \wedge, \neg, \forall, \exists$. We assume for simplicity that all formulæ are in Negation Normal Form (NNF). A variable x is *free* in ϕ if it occurs in ϕ , but not in the scope of the quantifier $\forall x$ or $\exists x$. If ϕ has no free variables then ϕ is *closed*.

An *interpretation* I maps every sort \mathbf{s} to a set of elements \mathbf{s}^I , every variable x of sort \mathbf{s} to an element $x^I \in \mathbf{s}^I$, every function symbol $f : \mathbf{s}_1 \times \dots \times \mathbf{s}_n \rightarrow \mathbf{s}$ to a function f^I from $\mathbf{s}_1^I \times \dots \times \mathbf{s}_n^I$ to \mathbf{s}^I and every defined symbol $d \in \mathcal{D}_{\mathbf{s}}$ to a subset of \mathbf{s}^I . The set $\bigcup_{\mathbf{s} \in \mathcal{S}} \mathbf{s}^I$ is the *domain* of I . As usual, any interpretation I can be extended to a function mapping every term t of sort \mathbf{s} to an element

$[t]^I \in \mathbf{s}^I$ and every formula ϕ to a truth value $[\phi]^I \in \{\text{true}, \text{false}\}$. We write $I \models \phi$ (and we say that I *validates* ϕ) if $[\forall \mathbf{x} \phi]^I = \text{true}$, where \mathbf{x} is the vector of free variables in ϕ . We assume, w.l.o.g., that the sets \mathbf{s}^I (for $\mathbf{s} \in \mathcal{S}$) are disjoint. Sets of formulæ are interpreted as conjunctions. If ϕ and ψ are two formulæ or sets of formulæ, we write $\phi \equiv_I \psi$ if either $I \models \phi$ and $I \models \psi$ or $I \not\models \phi$ and $I \not\models \psi$. We write $\phi \equiv \psi$ if $\phi \equiv_I \psi$ for all interpretations I .

We introduce two transformations operating on interpretations. The first one is simple: it only affects the value of some variables or constant symbols. If I is an interpretation, x_1, \dots, x_n are distinct variables or constant symbols of sort $\mathbf{s}_1, \dots, \mathbf{s}_n$ respectively and v_1, \dots, v_n are elements of $\mathbf{s}_1^I, \dots, \mathbf{s}_n^I$, then we denote by $I[v_1/x_1, \dots, v_n/x_n]$ the interpretation coinciding with I , except that for every $i = 1, \dots, n$, we have: $x_i^{I[v_1/x_1, \dots, v_n/x_n]} \stackrel{\text{def}}{=} v_i$.

The second transformation is slightly more complex. The idea is to change the values of the elements of an inductive sort, without affecting the remaining part of the interpretation. An \mathcal{I} -*mapping* for an interpretation I is a function λ mapping every element e in the domain of I to an element of the same sort, that is the identity on every element occurring in a set \mathbf{s}^I , where $\mathbf{s} \notin \mathcal{I}$. Then $\lambda(I)$ is the interpretation coinciding with I , except that for every symbol f of a sort $\mathbf{s} \notin \mathcal{I}$, we have: $f^{\lambda(I)}(e_1, \dots, e_n) \stackrel{\text{def}}{=} f^I(\lambda(e_1), \dots, \lambda(e_n))$.

In the following, we assume that all interpretations belong to a specific class \mathfrak{J} . This is useful to fix the semantics of some of the symbols, for instance one may assume that the interpretation of a sort `int` is not arbitrary but rather equal to \mathbb{Z} . Of course, \mathfrak{J} is not arbitrary: the following definitions specify all the conditions that must be satisfied by the considered class of interpretations. We start by the interpretation of the defined symbols. As explained in the Introduction, the value of these symbols are to be specified by convergent systems of rewriting rules, satisfying some additional conditions defined as follows:

Definition 2. *Let $<$ be an ordering on defined symbols. Let \mathfrak{R} be an orthogonal system of rules of the form $d_{f(x_1, \dots, x_n)} \rightarrow \phi$, where d is a defined symbol in \mathbf{s} , f is of profile $\mathbf{s}_1 \times \dots \times \mathbf{s}_n \rightarrow \mathbf{s}$, and x_1, \dots, x_n are distinct variables of sorts $\mathbf{s}_1, \dots, \mathbf{s}_n$. We assume that ϕ and \mathfrak{R} satisfy the following conditions:*

1. *The free variables of ϕ occur in x_1, \dots, x_n .*
2. *All \mathcal{I} -terms occurring in ϕ belong to the set $\{x_1, \dots, x_n, f(x_1, \dots, x_n)\}$.*
3. *If ϕ contains a formula d'_t then either $d' < d$ and $t = f(x_1, \dots, x_n)$, or $t \in \{x_1, \dots, x_n\}$.*
4. *For every constructor f , \mathfrak{R} contains a rule of the form $d_{f(x_1, \dots, x_n)} \rightarrow \phi$.*

It is clear from the conditions of Definition 2 that \mathfrak{R} is convergent (the condition on the ordering ensures termination, and orthogonality ensures confluence). We denote by $d_t \downarrow_{\mathfrak{R}}$ the normal form of d_t w.r.t. \mathfrak{R} . The following condition states that the interpretation of defined symbols must correspond to the one specified by the rewrite system \mathfrak{R} , for every interpretation in \mathfrak{J} .

Definition 3. *An interpretation is \mathfrak{R} -compatible iff for all sort symbols $\mathbf{s} \in \mathcal{I}$, for all defined symbols $d \in \mathcal{D}_{\mathbf{s}}$, for all function symbols $f : \mathbf{s}_1 \times \dots \times \mathbf{s}_n \rightarrow \mathbf{s}$, we have $d_{f(x_1, \dots, x_n)} \equiv_I d_{f(x_1, \dots, x_n)} \downarrow_{\mathfrak{R}}$.*

The second condition that is required ensures that any equation between two constructor terms can be reduced to equations between variables:

Definition 4. *An interpretation is \simeq -decomposable iff the following conditions hold:*

1. *For every $\mathbf{s} \in \mathcal{I}$ and for every $f, g \in \Sigma_{\mathbf{s}}$ of arity n and m respectively, there exists a formula $\Delta^{(f,g)}$ built on \vee, \wedge, \simeq and on $n+m$ distinct variables $x_1, \dots, x_n, y_1, \dots, y_m$ such that $f(x_1, \dots, x_n) \simeq g(y_1, \dots, y_m) \equiv_I \Delta^{(f,g)}$.*
2. *For every $i \in [1, n]$ we have $\Delta^{(f,g)} \models \bigvee_{k=1}^m x_i \simeq y_k$, and for every $j \in [1, m]$, we have $\Delta^{(f,g)} \models \bigvee_{k=1}^n y_j \simeq x_k$.*

If $t = f(t_1, \dots, t_n)$ and $s = g(s_1, \dots, s_m)$ are two non-variable \mathcal{I} -terms, we denote by $\Delta(t \simeq s)$ the formula obtained from $\Delta^{(f,g)}$ by replacing each variable x_i ($1 \leq i \leq n$) by t_i and each variable y_j ($1 \leq j \leq m$) by s_j .

Example 2. If, for instance, elements of a sort $\mathbf{s} \in \mathcal{I}$ are interpreted as terms built on a set of free constructors, then we have $\Delta^{(f,g)} \simeq \perp$ if $f \neq g$ and $\Delta^{(f,f)} \stackrel{\text{def}}{=} x_1 \simeq y_1 \wedge \dots \wedge x_n \simeq y_n$ (where n denotes the arity of f). Indeed, in this case, we have $f(x_1, \dots, x_n) \simeq f(y_1, \dots, y_n) \equiv (x_1 \simeq y_1 \wedge \dots \wedge x_n \simeq y_n)$. If, on the other hand, g is intended to denote a commutative binary function then we should have: $\Delta^{(g,g)} = (x_1 \simeq y_1 \wedge x_2 \simeq y_2) \vee (x_1 \simeq y_2 \wedge x_2 \simeq y_1)$. The variables x_i and y_j are those introduced in Definition 4.

The third condition ensures that the interpretation of every inductive sort is minimal (w.r.t. to set inclusion).

Definition 5. *An interpretation is \mathcal{I} -inductive iff for every $\mathbf{s} \in \mathcal{S}$, and for every element $u \in \mathbf{s}^I$, there exists a constructor term t such that $u = [t]^I$.*

Notice that, by definition, a constructor term contains no variable of a sort in \mathcal{I} . For instance, every element in \mathbf{nat}^I should be equal to a ground term $\text{succ}^k(0)$, for some $k \in \mathbb{N}$. If \mathbf{list} denotes the sort of the lists built on elements of a sort $\mathbf{s} \notin \mathcal{I}$, then any element of \mathbf{list}^I must be equal to a term of the form $\text{cons}(x_1, \text{cons}(x_2, \dots, \text{cons}(x_n, \text{nil}) \dots))$, where x_1, \dots, x_n are variables of sort \mathbf{s} . This condition implies in particular that for every $\mathbf{s} \notin \mathcal{I}$ and for every element $v \in \mathbf{s}^I$, there exists a variable x such that $x^I = v$ (this is obviously not restrictive, since the variables may be interpreted arbitrarily).

The next definition summarizes all the conditions that are imposed:

Definition 6. *A class of interpretations \mathfrak{I} is schematizable iff all interpretations $I \in \mathfrak{I}$ satisfy the following properties:*

1. *I is \mathfrak{R} -compatible.*
2. *I is \simeq -decomposable.*
3. *I is \mathcal{I} -inductive.*
4. *For all variables v of a sort \mathbf{s} and for all elements $e \in \mathbf{s}^I$, $I[e/v] \in \mathfrak{I}$.*
5. *For all \mathcal{I} -mappings λ , $\lambda(I) \in \mathfrak{I}$.*

A formula ϕ is \mathfrak{I} -satisfiable iff ϕ has a model in \mathfrak{I} .

From now on we focus on testing \mathcal{J} -satisfiability for a schematizable class of interpretations. Before that we impose some restrictions on the formulæ to be tested. As we shall see, these conditions will be useful mainly to ensure that the proof procedure presented in Section 3 only generates a finite number of distinct formulæ, up to a renaming of the parameters. This property is essential for the proof of termination, although it is not a sufficient condition.

Definition 7. *A class of formulæ \mathfrak{F} is admissible if all formulæ $\phi \in \mathfrak{F}$ satisfy the following properties:*

1. For all parameters A, B , $\phi[B/A] \in \mathfrak{F}$.
2. ϕ contains no constructor and no variable of a sort in \mathcal{I} .
3. For every subformula ψ of ϕ , if ψ is not a disjunction, a conjunction, or a defined atom, then ψ contains no defined symbol and no pairs of distinct parameters.
4. For every defined symbol d occurring in ϕ and for every rule $d_t \rightarrow \phi$ in \mathfrak{R} , the formula obtained from ϕ by replacing each \mathcal{I} -term by an arbitrary parameter is in \mathfrak{F} .

A formula occurring in \mathfrak{F} is a schema. It is a base formula iff it contains no defined symbol, and no equation between parameters.

The conditions in Definition 7 ensure that the formulæ in \mathfrak{F} are boolean combinations (built on \vee, \wedge) of base formulæ containing at most one parameter, of defined atoms and of equations and disequations between parameters. The definition of base formulæ in Definition 7 ensures that the truth values of base formulæ do not depend on the interpretation of the parameters, but only on the *relation* between them. Base formulæ can contain parameters, but they can only occur as arguments of function symbols, whose images must be of a non-inductive sort. The only way of specifying properties of the parameters themselves (and not of the terms built on them) is by using the rewrite rules in \mathfrak{R} . As we shall see, this property is essential for proving the soundness of the loop detection rule that ensures termination of our proof procedure. Similarly, no quantification over variables of an inductive sort is allowed.

In the following, \mathcal{J} denotes a schematizable class of interpretations and \mathfrak{F} denotes an admissible class of formulæ. The goal of the paper is to prove that if \mathcal{J} -satisfiability is decidable (resp. semi-decidable) for base formulæ in \mathfrak{F} then it must be so for all formulæ in \mathfrak{F} . We give examples of classes of formulæ satisfying the previous conditions:

Example 3. Assume that Σ only contains 0, succ and symbols of profile $\mathbf{nat} \rightarrow \mathbf{bool}$. Let \mathcal{J}_0 be the class of all \mathfrak{R} -compatible interpretations on this language with the usual interpretation of \mathbf{nat} , 0 and succ, and let \mathfrak{F}_0 be the set of all quantifier-free formulæ containing no occurrence of 0 and succ. Clearly, \mathcal{J}_0 is schematizable and \mathfrak{F}_0 is admissible. The formulæ in \mathfrak{F}_0 denote schemata of propositional formulæ. For instance the schema $p_0 \wedge \neg p_N \wedge \bigwedge_{K=0}^{N-1} (\neg p_K \vee p_{\text{succ}(K)})$ is specified by the formulæ: $p(0) \wedge \neg p(N) \wedge d_N$, where d is defined by the rules $d_0 \rightarrow \top$ and $d_{\text{succ}(K)} \rightarrow d_K \wedge (\neg p(K) \vee p(\text{succ}(K)))$. \mathfrak{F}_0 is equivalent to the class of *regular schemata* in [3].

Example 4. Let $\mathcal{S} = \{\mathbf{nat}, \mathbf{int}\}$ and $\mathcal{I} = \{\mathbf{nat}\}$. Assume that Σ contains the symbols 0 and succ, constant symbols of sort \mathbf{int} , function symbols of profile $\mathbf{nat} \rightarrow \mathbf{int}$ and all the symbols of Presburger arithmetic. Let $\mathcal{I}_{\mathbb{Z}}$ be the class of all \mathfrak{R} -compatible interpretations such that the interpretations of $\mathbf{nat}, \mathbf{int}, 0, \text{succ}, +, \leq, \dots$ are the usual ones. Let $\mathfrak{F}_{\mathbb{Z}}$ be the set of all formulæ built on this language, containing no occurrence of 0, succ, and satisfying Condition 3 in Definition 7. It can be easily checked that $\mathcal{I}_{\mathbb{Z}}$ is schematizable and that $\mathfrak{F}_{\mathbb{Z}}$ is admissible. Formulæ in $\mathfrak{F}_{\mathbb{Z}}$ denote schemata of Presburger formulæ (the base formulæ in $\mathfrak{F}_{\mathbb{Z}}$ are formulæ of Presburger arithmetic). For instance $\bigvee_{K=0}^N a(K) > 0$ is denoted by d_M , with the rules $d_0 \rightarrow (a(0) > 0)$ and $d_{\text{succ}(K)} \rightarrow d_K \vee a(\text{succ}(K)) > 0$. Note however, that schemata containing atoms with several distinct terms of sort \mathbf{nat} , such as $\bigwedge_{K=0}^N a(K) \simeq a(\text{succ}(K))$ cannot occur in $\mathfrak{F}_{\mathbb{Z}}$. It is also important to remark that the sort \mathbf{int} *must* be distinct from the sort of the indices \mathbf{nat} (terms of the form $d_{a(K)}$ are *not* allowed).

The class $\mathfrak{F}_{\mathbb{Z}}$ is not comparable to the class of SMT-schemata in 4 (the latter class may contain formulæ of the previous form, at the cost of additional restrictions on the considered theory). Let \mathcal{I}_1 and \mathfrak{F}_1 be the sets of interpretations and formulæ fulfilling the conditions of Definitions 6 and 7. The following proposition is easy to establish (\mathfrak{F}_0 and $\mathfrak{F}_{\mathbb{Z}}$ are defined in Examples 3 and 4):

Proposition 1. *\mathcal{I}_0 -satisfiability (resp. $\mathcal{I}_{\mathbb{Z}}$ -satisfiability) is decidable for base formulæ in \mathfrak{F}_0 (resp. $\mathfrak{F}_{\mathbb{Z}}$), and \mathcal{I}_1 -satisfiability is semi-decidable for base formulæ in \mathfrak{F}_1 .*

Before describing the proof procedure for testing the satisfiability of schemata, we provide a simple example of an application. It is only intended to give a taste of what can be expressed in our logic, and of which properties are outside its scope (see also the examples in the Introduction, that can be easily encoded).

Example 5. A (binary) DAG δ labeled by elements of type \mathbf{elem} can be denoted by a function symbol $\delta : \mathbf{DAG} \rightarrow \mathbf{elem}$, where the signature contains two constructors of sort \mathbf{DAG} : a constant symbol \perp (denoting the empty DAG), and a 3-ary symbol $c(n, l, r)$, where l and r denote the left and right children respectively and n denotes the current node². Various properties can be expressed in our logic, for instance the following defined symbol $A_x^{\delta,p}$ expresses the fact that all the elements occurring in a DAG δ satisfies some property p .

$$A_{\perp}^{\delta,p} \rightarrow \top \qquad A_{c(n,l,r)}^{\delta,p} \rightarrow A_l^{\delta,p} \wedge A_r^{\delta,p} \wedge p(\delta(c(n, l, r)))$$

Obviously this can be generalized to any set of regular positions: for instance, we can state that there exists a path from the root to a leaf in the DAG on which all the element satisfy p :

$$E_{\perp}^{\delta,p} \rightarrow \top \qquad E_{c(n,l,r)}^{\delta,p} \rightarrow (E_l^{\delta,p} \vee E_r^{\delta,p}) \wedge p(\delta(c(n, l, r)))$$

δ and p are meta-variables: δ must be replaced by a function symbol of profile $\mathbf{DAG} \rightarrow \mathbf{elem}$ and p can be replaced by any property of elements of sort \mathbf{elem} (provided it is expressible in the base language e.g. first-order logic). For instance, we can express the

² This extra-argument is necessary to ensure that distinct nodes can have the same children.

fact that all the elements of δ are equal to some fixed value, or that all the elements of δ are even. We can check that the following formula is valid: $(\forall x, p(x) \Rightarrow q(x)) \Rightarrow (E^{\delta,p} \Rightarrow E^{\delta,q})$. However, the converse *cannot* be expressed in our setting, because it would involve a quantification over an element of type **DAG** which is forbidden by Condition **2** in Definition **7**. The formula $A^{\delta,p} \wedge \neg A^{\delta,q} \wedge \neg A^{\delta,-q}$ is satisfiable on the interpretations whose domain contains two elements e_1, e_2 such that $p(e_1), p(e_2), \neg q(e_1)$, and $q(e_2)$ hold (but for instance it is unsatisfiable if $p(x) \equiv (x \simeq 0)$). We can express the fact that two DAGs δ and δ' share an element: $\exists x, \forall y, (p(y) \Leftrightarrow x = y) \wedge \neg A^{\delta,-p} \wedge \neg A^{\delta',-p}$. We can also define a symbol $\text{Map}^{\delta,\delta',f}$ stating that δ' is obtained from δ by applying some function f on every element of δ :

$$\text{Map}_{c(n,l,r)}^{\delta,\delta',f} \rightarrow \text{Map}_{\perp}^{\delta,\delta',f} \rightarrow \top$$

$$\text{Map}_{c(n,l,r)}^{\delta,\delta',f} \rightarrow \text{Map}_l^{\delta,\delta',f} \wedge \text{Map}_r^{\delta,\delta',f} \wedge \delta'(c(n,l,r)) = f(\delta(c(n,l,r)))$$

Then, we can check, for instance, that if all the elements of δ are even and if f is the successor function, then all the elements of δ' must be odd:

$$(even(0) \wedge (\forall x, even(succ(x)) \Leftrightarrow \neg even(x)) \wedge A_A^{\delta,even}) \wedge \text{Map}^{\delta,\delta',succ} \Rightarrow A_A^{\delta',-even}$$

We are not able, however, to express transformations affecting the *shape* of the DAG (e.g. switching all the right and left subgraphs) because this would require to use non-monadic defined symbols.

$\text{Alt}^{\delta,p,q}$ expresses the fact that all the elements at even positions satisfy p and that the elements at odd positions satisfy q :

$$\text{Alt}_{\perp}^{\delta,p,q} \rightarrow \top \quad \text{Alt}_{c(n,l,r)}^{\delta,p,q} \rightarrow \text{Alt}_l^{\delta,q,p} \wedge \text{Alt}_r^{\delta,q,p} \wedge p(\delta(c(n,l,r)))$$

Our procedure can be used to verify that $\text{Alt}_A^{\delta,p,q} \Rightarrow A_A^{\delta,p \vee q}$. The following defined symbol $p^{\delta,\delta',\delta''}$ states that a DAG δ'' is constructed by taking elements from δ and δ' alternatively:

$$p_{c(n,l,r)}^{\delta,\delta',\delta''} \rightarrow p_l^{\delta',\delta,\delta''} \wedge p_r^{\delta',\delta,\delta''} \wedge \delta''(c(n,l,r)) = \delta(c(n,l,r))$$

$$p_{\perp}^{\delta,\delta',\delta''} \rightarrow \top$$

We can check that if the elements of δ and δ' satisfy Properties p and q respectively, then the elements in δ'' satisfy p and q alternatively: $(p_A^{\delta,\delta',\delta''} \wedge A_A^{\delta,p} \wedge A_A^{\delta',q}) \Rightarrow \text{Alt}_A^{\delta'',p,q}$.

Notice that, in this example, the subgraphs can share elements. Thus it is not possible in general to reason independently on each branch (in the style of automata-based approaches): one has to reason *simultaneously* on the whole DAG. Other data structures such as arrays or lists can be handled in a similar way. An example of property that *cannot* be expressed is sortedness. Indeed, it would be stated as follows:

$$\text{Sort}_{c(n,l,r)}^{\delta} \rightarrow \text{Sort}_l^{\delta} \wedge \text{Sort}_r^A \wedge \delta(c(n,l,r)) \geq \delta_l \wedge \delta(c(n,l,r)) \geq \delta_r$$

However, the atom $\delta(c(n,l,r)) \geq \delta_l$ is *not* allowed in our setting: since it contains several parameters, it contradicts Condition **3** in Definition **7**.

3 Proof Procedure

In this section, we present our procedure for testing the \mathfrak{J} -satisfiability of admissible formulæ. We employ a tableaux-based procedure, with several kinds of

inference rules: *Decomposition rules* that reduce each formula to a conjunction of base formulæ, equational literals, and defined literals; *Unfolding rules* that allow to unfold the defined atoms (by applying the rules in \mathfrak{R}); *Equality rules* for reasoning on equational atoms; and *Delayed instantiation schemes* that replace a parameter A by some term $f(B_1, \dots, B_n)$, where f is a constructor and B_1, \dots, B_n are new constant symbols. We consider proof trees labeled by sets of formulæ. If α is a node in a tree \mathcal{T} then $\mathcal{T}(\alpha)$ denotes the label of α . A node is *closed* if it contains \perp . As usual, our procedure is specified by a set of *expansion rules* of the form

$$\frac{\Psi}{\Psi_1 \mid \dots \mid \Psi_n}$$

with $n \geq 1$, meaning that a non-closed leaf node labeled by a set $\Phi \supseteq \Psi$ (up to a substitution of the meta-variables) may be expanded by adding n children labeled by $(\Phi \setminus \Psi) \cup \Psi_1, \dots, (\Phi \setminus \Psi) \cup \Psi_n$ respectively. We assume moreover that the formulæ Ψ_1, \dots, Ψ_n have not already been generated in the considered branch (to avoid redundant applications of the rules). For any tree \mathcal{T} , we write $\alpha \geq_{\mathcal{T}} \beta$ iff β is a child of α . $\geq_{\mathcal{T}}^*$ denotes as usual the reflexive and transitive closure of $\geq_{\mathcal{T}}$.

We need to introduce some additional notations and definitions. For any interpretation I and for any element v in the domain of I , we denote by $\text{depth}_I(v)$ the depth of the constructor term denoted by v , formally defined as follows: $\text{depth}_I(v) = 0$ if v is in $D_{\mathfrak{s}}$ and $\mathfrak{s} \notin \mathcal{I}$, otherwise $\text{depth}_I([f(t_1, \dots, t_n)]^I) = 1 + \max(\{\text{depth}_I([t_i]^I) \mid i \in [1, n]\})$, with the convention that $\text{max}(\emptyset) = 0$. It is easy to check that the function $v \mapsto \text{depth}_I(v)$ is well-defined, for every interpretation $I \in \mathfrak{J}$.

For the sake of readability, we shall assume that there exists a function symbol *depth* such that: $\text{depth}^I(v) \stackrel{\text{def}}{=} \text{depth}_I(v)$. The formula $\text{max}(E) \simeq t$ (where E is a finite set of terms) is written as a shorthand for $\bigwedge_{s \in E}(s \leq t) \wedge \bigvee_{s \in E}(s \simeq t)$ if $E \neq \emptyset$ and for $0 \simeq t$ if $E = \emptyset$.

Let \mathcal{T} be a tree and let α be a node in \mathcal{T} . A parameter A is *solved* in α if the only formula of $\mathcal{T}(\alpha)$ containing A is of the form $A \simeq B$ where B is a parameter. An equation $A \simeq B$ is *solved* in α if A is solved. Notice that \simeq is *not* considered as commutative. For every set of formulæ Φ , $\text{Eq}(\Phi)$ denotes the set of equations in Φ and $\text{NonEq}(\Phi) \stackrel{\text{def}}{=} \Phi \setminus \text{Eq}(\Phi)$. A *renaming* is a function ρ mapping every parameter to a parameter of the same sort, such that $\rho(N) = N$. Any renaming ρ can be extended into a function mapping every formula ϕ to a formula $\rho(\phi)$, obtained by replacing every parameter A occurring in ϕ by $\rho(A)$. Let Φ and Ψ be two sets of formulæ. We write $\Phi \sqsupseteq \Psi$ iff there exists a renaming ρ such that $\rho(\Psi) \subseteq \Phi$.

A *proof tree* for ϕ is a tree constructed by the rules of Figure \square below and such that the root is obtained by applying START on ϕ . We assume that \vee -DECOMPOSITION and \wedge -DECOMPOSITION are applied with the highest priority.

Most of the rules in in Figure \square are self-explanatory. We only briefly comment on some important points.

START is only applied once, in order to create the root node of the tree. The label of this node contains the formula at hand together with an additional

formula stating that the max of the depth of the constructor terms represented by the parameters must equal to some natural number N .

The decomposition and closure rules are standard. However, we do *not* use them to test the satisfiability of the formula, but only to decompose it into a conjunction of defined atoms, equational literals and base formulæ. This is always feasible, thanks to the particular properties of formulæ in \mathfrak{F} (see Definition 7). Notice that the separation rule has no premises. The only requirement is that A and B occur in the considered branch.

UNFOLDING replaces a defined atom d_A by its definition according to the rules in \mathfrak{R} . This is possible only when the head symbol and arguments of the term represented by A are known.

\simeq -DECOMPOSITION decomposes equalities, using the specific properties of \simeq -decomposable interpretations: if a node contains two equations $A \simeq t$ and $A \simeq s$ then the formula $\Delta(t \simeq s)$ necessarily holds. $\not\simeq$ -DECOMPOSITION performs a similar task for inequalities.

Several rules are introduced to reason on the depth of the terms represented by the parameters. The principle is to separate the parameters representing terms of a depth exactly equal to N from those whose depth is strictly less than N (so that only the former ones may be instantiated). By definition of START, the initial node must contain an equation $depth(A) \preceq N$ for each parameter $A \neq N$. STRICTNESS expands this inequality by using the equivalence $x \preceq y \Leftrightarrow (x \prec y \vee x \simeq y)$. Then \vee -DECOMPOSITION will apply, yielding either $x \prec y$ or $x \simeq y$. \prec -DECOMPOSITION gets rid of strict equalities of the form $depth(A) \prec succ(t)$ that are introduced by N -EXPLOSION.

The Explosion rules instantiate the parameters, which is done by adding equations of the form $A \simeq f(\mathbf{B})$, where \mathbf{B} is a vector of fresh parameters.

EXPLOSION instantiates the parameters distinct from N . We choose to instantiate only the parameters representing terms of maximal depth, and only after N has been instantiated. Thus we instantiate a parameter B only if there exists an atom of the form $depth(B) \simeq t$, where t is of the form $succ(s)$, for some $s \in \{0, N\}$. EXPLOSION enables further applications of UNFOLDING, which in turn may introduce new complex formulæ into the nodes (by unfolding the defined symbols according to the rules in \mathfrak{R}).

N -EXPLOSION instantiates the parameter N . Since the depth of the terms of a sort in \mathcal{I} is at least 1 and since N is intended to denote the maximal depth of the parameters, N cannot be 0, thus it is instantiated either by $succ(0)$ or by $succ(N)$. Unlike the other parameters, direct replacement is performed. This rule is applied with the lowest priority. Hence, when the rule is applied, all parameters of a depth strictly greater than N must have been instantiated. By replacing N by a term of the form $succ(t)$, the rule will permit to instantiate the parameters of depth $N - 1$. This strategy ensures that the parameters will be instantiated in decreasing order w.r.t. the depth of the terms they represent.

LOOP is intended to detect cycles and prune the corresponding branches, by closing the nodes that are subsumed by a previous one. It only applies on some particular nodes, that are irreducible w.r.t. all rules, except (possibly)

N-EXPLOSION. We shall call any such node a *layer*. This rule can be viewed as an application of the induction principle. If $\Phi \sqsubseteq \Psi$ then it is clear that Ψ is a logical consequence of Φ , up to a renaming of parameters. Thus, if some open node exists below a node labeled by Φ , some other open node must exist also below a node labeled by Ψ , hence the node corresponding to Φ may be closed without threatening soundness (a satisfiable branch is closed, but global satisfiability is preserved). Since Ψ is a layer, the parameter N must be instantiated at least once between the two nodes, which ensures that the reasoning is well-founded and that there exists at least one open node outside the branch of Φ .

At first glance, it may seem odd to remove equations from Φ and Ψ before testing for subsumption (see the application condition of LOOP). Indeed, it is clear that this operation does *not* preserve satisfiability in general. For instance, the formula $p(A) \wedge \neg p(B) \wedge d_B \wedge A \simeq 0$ is unsatisfiable if d is defined by the rules: $d_0 \rightarrow \top$ and $d_{\text{succ}(K)} \rightarrow \perp$. However, $p(A) \wedge \neg p(B) \wedge d_B$ is satisfiable (with $A^I \neq 0$). In the context in which the rule is applied however, it will be ensured that satisfiability is preserved. The intuition is that if an equation such as $A \simeq 0$ occurs in the node, then A must have been instantiated previously, hence the term represented by A must be of a depth strictly greater than N . Due to the chosen instantiation strategy, all parameters of depth greater or equal to that of A , must have been instantiated (this property is not fulfilled by the previous formula: B should be instantiated since its depth is at most 1 by definition). Then it may be seen that the interpretation of the remaining formulæ does not depend on the value of A , since the depth of their indices must be strictly less than that of A . Note that the removal of equations is *essential* for ensuring termination.

We provide a simple example to illustrate the rule applications.

Example 6. Consider the formula $\forall x \neg p(x) \wedge d_A$, together with the rules: $d_a \rightarrow p(b)$ and $d_{f(x,y)} \rightarrow d_x \wedge d_y$ (where $\mathcal{C} = \{a:s, f:s \times s \rightarrow s, 0, \text{succ}\}$ and $\text{profile}(A) = \mathbf{s}$). The root formula is $\forall x \neg p(x) \wedge d_A \wedge \max(\{\text{depth}(A)\}) \simeq N$. By normalization using \wedge -DECOMPOSITION we get $\{\forall x \neg p(x), d_A, \text{depth}(A) \simeq N\}$. No rule applies, except *N*-EXPLOSION, which replaces N by $\text{succ}(0)$ or $\text{succ}(N)$. In both cases, EXPLOSION applies on A . In the first branch, the rule adds the formula $A \simeq a$ and in the second one, it yields $A \simeq f(B, C)$ (where B, C are fresh parameters). In the former branch, UNFOLDING replaces the formula d_A by $p(b)$, then an irreducible node is reached. In the latter branch, the formulæ d_B and d_C are inferred. Then LOOP applies, using the renaming: $\rho(A) = B$ or $\rho(A) = C$, hence the node is closed. The only remaining (irreducible) node is $\{p(b), \forall x \neg p(x)\}$. The unsatisfiability of this set of formulæ can be easily checked.

The following example shows evidence of the importance of the depth rules:

Example 7. Consider the formula: $p(A) \wedge d_A \wedge c_B$ with the rules $d_{\text{succ}(x)} \rightarrow d_x, d_0 \rightarrow \top, c_{\text{succ}(x)} \rightarrow \perp$ and $c_0 \rightarrow \neg p(0)$. If the parameters were instantiated in an arbitrary order, then one could choose for instance to instantiate A by $\text{succ}(A')$, yielding an obvious loop (indeed, the unfolding of d_A yields $d_{A'}$, thus it suffices to consider the renaming $\rho(A) = A'$ and $\rho(B) = B$). Then the only remaining branch corresponds to the case $A \simeq 0$, which is actually unsatisfiable. This trivial but instructive example shows that reasoning on the depth of the parameters is necessary to ensure that the model will

START: $\frac{\phi, \max(\{depth(A_i) \mid i \in [1, n]\}) \simeq N}{\phi, \max(\{depth(A_i) \mid i \in [1, n]\}) \simeq N}$		Where ϕ denotes the formula at hand A_1, \dots, A_n are the parameters in ϕ
V-DECOMPOSITION: $\frac{\phi \vee \psi}{\phi \mid \psi}$		\wedge -DECOMPOSITION: $\frac{\phi \wedge \psi}{\phi, \psi}$
CLOSURE: $\frac{\neg\phi, \phi}{\perp}$		\simeq -CLOSURE: $\frac{A \not\approx A}{\perp}$
UNFOLDING: $\frac{d_A, A \simeq f(\mathbf{B})}{\psi}$		$\frac{-d_A, A \simeq f(\mathbf{B})}{NNF(\neg\psi)}$
$f(\mathbf{B})$		$\psi = d_{f(\mathbf{B})} \downarrow_{\mathfrak{R}} [A/f(\mathbf{B})], A \simeq f(\mathbf{B})$
\simeq -DECOMPOSITION: $\frac{A \simeq f(\mathbf{B}), A \simeq g(\mathbf{C})}{\psi, A \simeq f(\mathbf{B})}$		$\frac{A \not\approx B, A \simeq f(\mathbf{B}), B \simeq g(\mathbf{C})}{NNF(\neg\psi), A \not\approx B, A \simeq f(\mathbf{B}), B \simeq g(\mathbf{C})}$
Where $\psi = \Delta(f(\mathbf{B}) \simeq g(\mathbf{C}))^a$		
REPLACEMENT: $\frac{\phi, A \simeq B}{\phi[B/A], A \simeq B}$		If A and B are two parameters and A occurs in ϕ
STRICTNESS: $\frac{depth(A) \preceq N}{depth(A) \simeq N \vee depth(A) \prec N}$		\prec -DECOMPOSITION: $\frac{t \prec succ(N)}{t \preceq N}$
\prec -SEPARATION: $\frac{depth(A) \prec N, depth(B) \simeq N}{depth(A) \prec N, depth(B) \simeq N, A \not\approx B}$		SEPARATION: $\frac{}{A \simeq B \vee A \not\approx B}$
EXPLOSION: $\frac{depth(B) \simeq succ(t)}{\bigvee_{i \in [1, n]} \max(E_i) \simeq t \wedge B \simeq t_i}$		
<p style="text-align: center;"> If t_i are terms of the form $f_i(\mathbf{A}_i)$, such that f_1, \dots, f_n are all the function symbols of the same sort as B, and the \mathbf{A}_i's are vectors of pairwise distinct, fresh, constant symbols of the appropriate sort, and E_i is the set of terms $depth(C)$, where C is a component of \mathbf{A}_i of a sort in \mathcal{I}. </p>		
N-EXPLOSION: $\frac{\Phi}{\Phi[succ(0)/N] \mid \Phi[succ(N)/N]}$		
<p style="text-align: center;"> If no other rule applies and N occurs in Φ. Notice that in contrast with the previous rules, Φ must denote the whole label (not a subset of it) </p>		
LOOP: $\frac{\Phi}{\perp}$		If there exists in the same branch a (non leaf) layer labeled by a set of formulæ Ψ such that $\text{NonEq}(\Phi) \sqsupseteq \text{NonEq}(\Psi)$
^a See Definition 4 for the definition of $\Delta(t \simeq s)$		

Fig. 1. Expansion rules

eventually be reached. In this example, the depth of A is maximal and that of B is not, e.g.: $A \simeq \text{succ}(0)$ and $B \simeq 0$. The problem stems from the fact that LOOP is *not* sound in general, since equational atoms are removed from the formulæ before testing for subsumption (the removal of such atoms is *crucial* for termination).

4 Properties of the Proof Procedure

This short section merely contains the theorems formalizing the main properties of the proof procedure. Due to space restrictions, the proofs are omitted and can be found in [13]. We first state that the previous rules are sound.

Theorem 1. *Let \mathcal{T} be a proof tree for a formula ϕ . If \mathcal{T} is closed then ϕ is unsatisfiable.*

We then state that the procedure is complete, in the sense that the satisfiability of every irreducible node can be tested by the procedure for base formulæ.

Theorem 2. *Let \mathcal{T} be a proof tree. If α is a node in \mathcal{T} that is irreducible by all the expansion rules then $\mathcal{T}(\alpha)$ is \mathfrak{I} -satisfiable iff $\text{NonEq}(\mathcal{T}(\alpha))$ is. Furthermore, $\text{NonEq}(\mathcal{T}(\alpha))$ is a set of base formulæ.*

We finally state that the procedure is terminating.

Theorem 3. *The expansion rules terminate on every formula in \mathfrak{F} .*

Corollary 1. *If the satisfiability problem is decidable (resp. semi-decidable) for base formulæ in \mathfrak{F} then it is so for all formulæ in \mathfrak{F} .*

5 Conclusion

We have proposed a proof procedure for reasoning on schemata of formulæ (defined by induction on an arbitrary structure, such as natural numbers, lists, trees etc.) by relating the satisfiability problem for such schemata to that of a *finite* disjunction of formulæ in the base language. Our approach applies to a wide range of formulæ, which may be interpreted in some specific class of structures (e.g. arithmetics). It may be seen as a generic way to add inductive capabilities into logical languages, in such a way that the main computational properties of the initial language (namely decidability or semi-decidability) are preserved. To the best of our knowledge, no published procedure offers similar features. There are very few decidability or even completeness results in inductive theorem proving and we hope that the present work will help to promote new progress in this direction. Future work includes the implementation of the proof procedure and its extension to non-monadic defined symbols.

References

1. Aczel, P.: An Introduction to Inductive Definitions. In: Barwise, K.J. (ed.) *Handbook of Mathematical Logic*, pp. 739–782. North-Holland, Amsterdam (1977)
2. Aravantinos, V., Caferra, R., Peltier, N.: A Schemata Calculus for Propositional Logic. In: Giese, M., Waaler, A. (eds.) *TABLEAUX 2009*. LNCS, vol. 5607, pp. 32–46. Springer, Heidelberg (2009)
3. Aravantinos, V., Caferra, R., Peltier, N.: Decidability and undecidability results for propositional schemata. *Journal of Artificial Intelligence Research* 40, 599–656 (2011)
4. Aravantinos, V., Peltier, N.: Schemata of SMT-Problems. In: Brünnler, K., Metcalfe, G. (eds.) *TABLEAUX 2011*. LNCS, vol. 6793, pp. 27–42. Springer, Heidelberg (2011)
5. Baaz, M., Hetzl, S., Leitsch, A., Richter, C., Spohr, H.: CERES: An analysis of Fürstenberg’s proof of the infinity of primes. *Theor. Comput. Sci.* 403(2-3), 160–175 (2008)
6. Baelde, D., Miller, D., Snow, Z.: Focused Inductive Theorem Proving. In: Giesl, J., Hähnle, R. (eds.) *IJCAR 2010*. LNCS, vol. 6173, pp. 278–292. Springer, Heidelberg (2010)
7. Benzmüller, C., Paulson, L.C., Theiss, F., Fietzke, A.: LEO-II - A Cooperative Automatic Theorem Prover for Classical Higher-Order Logic (System Description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) *IJCAR 2008*. LNCS (LNAI), vol. 5195, pp. 162–170. Springer, Heidelberg (2008)
8. Bouhoula, A., Kounalis, E., Rusinowitch, M.: SPIKE, an Automatic Theorem Prover. In: Voronkov, A. (ed.) *LPAR 1992*. LNCS, vol. 624, pp. 460–462. Springer, Heidelberg (1992)
9. Bouhoula, A., Rusinowitch, M.: Implicit induction in conditional theories. *Journal of Automated Reasoning* 14, 14–189 (1995)
10. Boyer, R.S., Moore, J.S.: A Theorem Prover for a Computational Logic. In: Stickel, M.E. (ed.) *CADE 1990*. LNCS, vol. 449, pp. 1–15. Springer, Heidelberg (1990)
11. Bundy, A.: The automation of proof by mathematical induction. In: Robinson, J.A., Voronkov, A. (eds.) *Handbook of Automated Reasoning*, pp. 845–911. Elsevier and MIT Press (2001)
12. Comon, H.: Inductionless induction. In: Robinson, A., Voronkov, A. (eds.) *Handbook of Automated Reasoning*, ch. 14, pp. 913–962. North-Holland (2001)
13. Echenim, M., Peltier, N.: Reasoning on Schemata of Formulae. Technical report, CoRR, abs/1204.2990 (2012)
14. Giesl, J., Kapur, D.: Decidable Classes of Inductive Theorems. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) *IJCAR 2001*. LNCS (LNAI), vol. 2083, pp. 469–484. Springer, Heidelberg (2001)
15. Gupta, A., Fisher, A.L.: Parametric Circuit Representation Using Inductive Boolean Functions. In: Courcoubetis, C. (ed.) *CAV 1993*. LNCS, vol. 697, pp. 15–28. Springer, Heidelberg (1993)
16. Kapur, D., Musser, D.: Proof by consistency. *Artificial Intelligence* 31 (1987)
17. Lenzi, G.: A New Logical Characterization of Büchi Automata. In: Ferreira, A., Reichel, H. (eds.) *STACS 2001*. LNCS, vol. 2010, pp. 467–477. Springer, Heidelberg (2001)
18. Park, D.M.: Finiteness is Mu-ineffable. *Theoretical Computer Science* 3, 173–181 (1976)
19. Paulin-Mohring, C.: Inductive Definitions in the system Coq - Rules and Properties. In: Bezem, M., Groote, J.F. (eds.) *TLCA 1993*. LNCS, vol. 664, pp. 328–345. Springer, Heidelberg (1993)

Management of Change in Declarative Languages

Mihnea Iancu and Florian Rabe

Jacobs University, Bremen, Germany

Abstract. Due to the high degree of interconnectedness of formal mathematical statements and theories, human authors often have difficulties anticipating and tracking the effects of a change in large bodies of symbolic mathematical knowledge. Therefore, the automation of change management is often desirable. But while computers can in principle detect and propagate changes automatically, this process must take the semantics of the underlying mathematical formalism into account. Therefore, concrete management of change solutions are difficult to realize.

The MMT language was designed as a generic declarative language that captures universal structural features while avoiding a commitment to a particular formalism. Therefore, it provides a promising framework for the systematic study of changes in declarative languages. We leverage this framework by providing a generic change management solution at the MMT level, which can be instantiated for arbitrary specific languages.

1 Introduction

Mathematical knowledge is growing at an enormous rate. Even if we restrict attention to formalized mathematics, libraries are reaching sizes that users have difficulties overseeing. Since this knowledge is also highly interconnected, it is getting increasingly difficult for humans to anticipate and follow the effects of changes. Therefore, management of change (MoC) for mathematics has received attention recently.

In this paper, we focus on change management for formalized mathematics, which — contrary to traditional, semi-formal mathematics — permits mechanically computing and verifying declarations. In principle, this should permit change management tools to automatically identify and recheck those declarations that are affected by a change. However, current computer algebra and deduction systems have not been designed systematically with change management in mind. In fact, the question of how to do that is still open.

A major motivation of our work was to provide change management for the LATIN library [CHK⁺11], a collection of formalizations of logics and related languages in a logical framework. Using the Little Theories approach [FGT92], the LATIN library takes the form of a highly modular and inter-connected network of theories, which creates an urgent need for change management.

We contribute to the solution of this problem by studying change management for the MMT language [RK11]. Because it was introduced as a foundation-

independent, modular, and scalable representation language for formal mathematical knowledge, it is a very promising framework for change management. Firstly, **foundation-independence** means that MMT avoids a syntactic or semantic commitment to any particular formalism. Thus, an MMT-based change management system could be applied to virtually any formal system. Secondly, **modularity** is a well-known strategy to rein in the impacts of changes and has been the basis of successful change management solutions such as [AHMS02]. Thirdly, MMT deemphasizes sequential in-memory processing of declarations in favor of maintaining a **large scale** network of declarations that are retrieved on demand, a crucial prerequisite for revisiting exactly the affected declarations.

We introduce a formal notion of differences between MMT documents, an abstract notion of semantic dependency relation, and a change propagation algorithm that guarantees that validity is preserved. We state our results for a small fragment of MMT, but our treatment extends to the full language. Our solution is implemented within the MMT system [Rab08], thus providing a generic change management system for formal mathematical languages.

In Sect. 2, we briefly introduce the MMT language in order to be self-contained. In Sect. 3, we refine our problem statement and compare it to related work. Then we develop the theory of change management in MMT in Sect. 4 and give an overview of our implementation in Sect. 5.

2 The MMT Language

We will only give a brief overview of MMT and refer to [RK11] for details. The fragment of the MMT grammar that we discuss in this paper is given in Fig. 1. In particular, we have to omit the MMT module system for simplicity. The central notion is that of a **theory graph**, a list of modules, which are theories T or theory morphisms v .

A **theory** declaration $T = \{Sym^*\}$ introduces a theory with name T containing a list of symbol declarations. A **symbol** declaration $c : \omega = \omega'$ introduces a symbol named c with **type** ω and **definiens** ω' . Both type and definiens are

Theory Graph	\mathcal{G}	$::= \cdot \mid \mathcal{G}, Mod$
Module Declaration	Mod	$::= T = \{Sym^*\} \mid v : T \rightarrow T = \{Ass^*\}$
Symbol Declaration	Sym	$::= c : \omega = \omega$
Assignment Declaration	Ass	$::= c := \omega$
Term	ω	$::= \perp \mid T?c \mid x \mid \omega\omega^+ \mid \omega X.\omega \mid \omega^v$
Variable Context	X	$::= \cdot \mid X, x : \omega = \omega$
Module Identifier	M	$::= T \mid v$
Theory Identifier	T	$::= \text{MMT URI}$
Morphism Identifier	v	$::= \text{MMT URI}$
Local Declaration Name	c	$::= \text{MMT Name}$

Fig. 1. Simplified MMT Grammar

optional. However, in order to reduce the number of case distinctions, we use the special term \perp : If the type or definiens is omitted, we assume they are \perp .

Terms ω over a theory T are formed from constants $T?c$ declared in T , bound variables x , application $\omega \omega_1 \dots \omega_n$ of a function ω to a sequence of arguments, bindings $\omega X.\omega'$ using a binder ω , a bound variable context X , and a scope ω' , and morphism application ω^v . Except for morphism application, this is a fragment of the OPENMATH language [BCC⁺04], which can express virtually every object.

Theory morphism declarations $v : T \rightarrow T' = \{Ass^*\}$ introduce a morphism with name v from T to T' containing a list of assignment declarations. Such a morphism must contain exactly one assignment $c := \omega'$ for each undefined symbol $c : \omega = \perp$ in T ; here ω' is some term over T' . Theory morphisms extend homomorphically to a mapping of T -terms to T' terms.

Intuitively, a theory morphism formalizes a translation between two formal languages. For example, the inclusion from the theory of semigroups to the theory of monoids (which extends the former with two declarations for the unit element and the neutrality axiom) can be formalized as a theory morphism. More complex examples are the Gödel-Gentzen negative translation from classical to intuitionistic logic or the interpretation of higher-order logic in set theory.

Every MMT declaration is identified by a canonical, globally unique URI. In particular, the URIs of symbol and assignment declarations are of the form $T?c$ and $v?c$.

MMT symbol declarations subsume most semantically relevant statements in declarative mathematical languages including function and predicate symbols, type and universe symbols, and — using the Curry-Howard correspondence — axioms, theorems, and inference rules. Their syntax and semantics is determined by the foundation, in which MMT is parametric. In particular, the validity of a theory graph is defined relative to a type system provided by the **foundation**:

Definition 1. A *foundation* provides for every theory graph \mathcal{G} a binary relation on terms that is preserved under morphism application. This relation is denoted by $\mathcal{G} \vdash \omega : \omega'$, i.e., we have $\mathcal{G} \vdash \omega : \omega'$ implies $\mathcal{G} \vdash \omega^v : \omega'^v$.

Constant declarations $c : \omega = \omega'$ in a theory graph \mathcal{G} are valid if $\mathcal{G} \vdash \omega' : \omega$. Thus, a foundation also has to define typing for the special term \perp : The judgment $\mathcal{G} \vdash \perp : \omega$ is interpreted as “ ω is a well-typed universe, i.e., it is legal to declare constants with type ω ”. Similarly, $\mathcal{G} \vdash \omega : \perp$ means that ω may occur as the definiens of an untyped constant. This way the foundation can precisely control what symbol declarations are well-formed. Similarly, an assignment $c := \omega$ in a morphism v is valid if $\mathcal{G} \vdash \omega : \omega'^v$ where ω' is the type of c in the domain of v .

Running Example 1. Below we present a simple MMT theory for propositional logic over two revisions Rev_1 and Rev_2 . For simplicity, we will assume that the MMT module system is used and that the symbols **type**, \rightarrow , and λ have been imported from a theory representing the logical framework LF, and that all theory graphs are validated relative to a fixed foundation for LF. PL of Rev_1 introduces a type **bool** of formulas and three binary connectives, the last of which is

defined in terms of the other two. This theory is valid. In Rev_2 , $bool$ is renamed to $form$, \vee is deleted, and \neg is added. The other declarations remain unchanged, thus making the theory invalid.

$$\begin{array}{ll}
 Rev_1 & Rev_2 \\
 PL = \{ & PL = \{ \\
 \quad bool : \mathbf{type} = \perp & \quad form : \mathbf{type} = \perp \\
 \quad \vee : bool \rightarrow bool \rightarrow bool = \perp & \quad \neg : form \rightarrow form = \perp \\
 \quad \wedge : bool \rightarrow bool \rightarrow bool = \perp & \quad \wedge : bool \rightarrow bool \rightarrow bool = \perp \\
 \quad \Rightarrow : bool \rightarrow bool \rightarrow bool & \quad \Rightarrow : bool \rightarrow bool \rightarrow bool \\
 \quad \quad = \lambda x.\lambda y.y \vee (x \wedge y) & \quad = \lambda x.\lambda y.y \vee (x \wedge y) \\
 \} & \}
 \end{array}$$

3 Related Work

MoC has been applied successfully in a number of **domains** such as software engineering (e.g., [EG89]) or file systems ([Apa00],[CVS],[Git]). A typical MoC work flow in this setting uses *compilation units*, e.g., the classes of a Java program: These are compiled independently, and a compilation manager can record the dependency relation between the units. In particular, if compilation units correspond to source files, changes in a file can be managed by recompiling all depending source files.

Intuitively, this work flow can be applied to declarative languages for mathematics as well if we replace “compilation” with “validation” where the latter includes, e.g., type reconstruction, rewriting, and theorem proving. However, there are a few key differences. Firstly, the validation units are individual types and definitions (which includes assertions and proofs in MMT), of which there are many per source file (around 50 on average in the Mizar library [TB85]). Their validation can be expensive, and there may be many dependencies within the same theory and many little theories in the same source file. Therefore, validation units cannot be mapped to files so that the notions of change and dependency must consider fragments of source files. Moreover, since foundations may employ search with backtracking, the validation of a unit U may access more units than the validity of U depends on. Therefore, the dependency relation should not be recorded by a generic MMT validation manager but produced by the foundation. Recently several systems have become able to produce such dependency relations, in particular Coq and Mizar [AMU11].

MoC systems for mathematical languages can be classified according to the **nature** of changes. In principle, any change in a declarative language can be expressed as a sequence of *add* and *delete* operations on declarations. But using additional change natures is important for scalability. We use *updates* to change the type or definiens of a declaration without changing its MMT URI, and *renames* to change only the MMT URI. We do not use *reordering* operations because MMT already guarantees that the order of declarations has no effect on the semantics. More complex natures have been studied in [BC08], which uses

splits in ontologies to replace one concept with two new ones. Dually, we could consider *merge* changes, which identify two declarations.

Moreover, MoC systems can be classified by the **abstraction level** of their document model. The most concrete physical and bit level are relatively boring, and standard MoC tools operate at the character level treating documents as arrays of lines [Apa00, CVS, Git]. More abstract document models such as XML are better suited for mathematical content [AM10, Wag10] and have been applied to document formats for mathematics [Wag10, ADD⁺11]. Our work continues this development to more abstract document models by using MMT, which specifically models mathematical data structures. For example, the order of declarations, the flattening of imports, and the resolution of relative identifiers are opaque in XML but transparent in MMT representations. Moreover, MMT URIs are more suitable to identify the validation units than the XPath-based URLs usually employed in generic XML-based change models.

The development graph model [AHMS99], which has been applied to change management in [Hut00, AHMS02], is very similar to MMT: Both are parametric in the underlying formal language, and both make the modular structure of mathematical theories explicit. The main difference is that MMT uses a concrete (but still generic) declarative language for mathematical theories; modular structure is represented using special declarations. Somewhat dually, development graphs use an abstract category of theories using diagrams to represent modular structure; the declarations within a theory can be represented by refining the abstract model as done in [AHM10].

At an even more abstract level, document models can be specific to one foundational language. While foundation-independent approaches like ours can only identify potentially affected validation units, those could determine and possibly repair the impact of a change. That would permit treating even subobjects as validation units. However, presently no systems exists that can provide such foundation-specific information so that such MoC systems remain future work.

Finally we can classify systems based on how they **propagate** changes. Our approach focuses on the theoretical aspect of identifying the (potentially) affected parts. The most natural post-processing steps are to revalidate them, as, e.g., in [AHMS02], or to present them for user interaction as in [ADD⁺11]. The MMT system can be easily adapted for either one. A very different treatment is advocated in [KK11] based on using only references that include revision numbers so that changes never affect other declarations (because each change generates a new revision number).

4 A Theory of Changes

4.1 A Data Structure for Changes

Just like we can consider only an exemplary fragment of MMT here, we can only consider some of the possible changes. We will only treat changes of declarations within modules. This is justified because these occur most frequently. However,

our treatment can be generalized to changes of any declaration in the full MMT language. The grammar for our formal **language of changes** is given in Fig. 2.

Diff	Δ	$::= \cdot \mid \Delta \bullet \delta$	
Change	δ	$::= \mathcal{A}(M, c : \omega = \omega) \mid \mathcal{D}(M, c : \omega = \omega) \mid \mathcal{U}(M, c, o, \omega, \omega') \mid \mathcal{R}(T, c, c)$	
Component	o	$::= \mathbf{def} \mid \mathbf{tp}$	
Box Terms	ω	$::= \boxed{\omega} \mid \boxed{\cdot}$	in addition to existing productions for ω

Fig. 2. The Grammar for MMT Changes

We use terms as validation units because they are the smallest units that can be validated separately by foundations. Therefore, besides adding and deleting whole declarations, we use updates that change a term. In updates, we use **components** o to distinguish between changes to the type ($o = \mathbf{tp}$) or the definiens ($o = \mathbf{def}$). More precisely:

- $\mathcal{A}(M, c : \omega = \omega')$ **adds** a declaration to the module M
- $\mathcal{D}(M, c : \omega = \omega')$ **deletes** a declaration from the module M .
- $\mathcal{U}(M, c, o, \omega, \omega')$ **updates** component o of declaration $M?c$ from ω to ω' .
- $\mathcal{R}(T, c, c')$ **renames** the declaration c in theory T to c' .

Finally, **Diffs** Δ are sequences of changes. In our implementation, we locate changes even more precisely by referring to subobjects of type and definiens. This is important for user interaction: If an impact has been detected, this permits showing the user exactly what change caused the impact.

Notation 1. In order to unify the cases of changing symbols in a theory and assignments in a morphism, we use the following convention: A declaration $c := \omega'$ in a morphism v abbreviates a declaration $c : \omega = \omega'$ and the components \mathbf{tp} and \mathbf{def} are defined accordingly. The type ω is uniquely determined by MMT to ensure the type preservation of morphisms: Its value is τ^v where τ is the type of c in the domain of v . Updates to assignments work in the same way as updates to symbols except that the component \mathbf{tp} cannot be changed.

We need one additional detail in our grammar: We add two special productions for terms ω , which we call **box terms**. These represent invalid terms that are introduced during change propagation.

$\boxed{\cdot}$ represents a missing term. $\boxed{\omega}$ represents a possibly invalid term ω . More sophisticated box terms can also record the required type, which gives users a hint what change is needed and permits applications to type-check a declaration relative to the box terms in it. We omit this here for simplicity.

Algebraically, the set of diffs Δ is the free monoid generated from changes δ . As we will see below, the operation of applying a diff to a theory graph can be regarded as this monoid acting on the set of theory graphs.

As seen on the right, our diffs are **invertible**. This permits transactions where partially applied diffs are rolled back if they cause an error. This is also useful to offer undo-redo functionality in a user interface.

In order to talk efficiently about MMT theory graphs, we introduce a few definitions that permit looking up information in the theory graph:

\cdot^{-1}	$= \cdot$
$(\Delta \bullet \delta)^{-1}$	$= \delta^{-1} \bullet \Delta^{-1}$
$\mathcal{A}(M, c : \omega = \omega')^{-1}$	$= \mathcal{D}(M, c : \omega = \omega')$
$\mathcal{D}(M, c : \omega = \omega')^{-1}$	$= \mathcal{A}(M, c : \omega = \omega')$
$\mathcal{R}(T, c, c')^{-1}$	$= \mathcal{R}(T, c', c)$
$\mathcal{U}(M, c, o, \omega, \omega')^{-1}$	$= \mathcal{U}(M, c, o, \omega', \omega)$

Definition 2 (Lookup in Theory Graphs). For a theory graph \mathcal{G} , we write

- $\vdash \mathcal{G}(M) = Mod$ if a module declaration Mod with URI M is present in \mathcal{G} .
- $\mathcal{G} \vdash T?c : \omega = \omega'$ if T is a theory URI in \mathcal{G} and the symbol declaration $c : \omega = \omega'$ exists in the body of T . We also define the corresponding notation for morphisms.
- $\vdash \mathcal{G}(M?c) = Sym$ if $\vdash \mathcal{G}(M) = Mod$ and Sym is the declaration with name c in the body of Mod .
- $\vdash \mathcal{G}(M?c/o) = \omega$ if $\vdash \mathcal{G}(M?c) = Sym$ and ω is the component o of Sym .
- $\mathcal{G} \vdash \pi$ if $\vdash \mathcal{G}(\pi) = Dec$ for some module or symbol declaration Dec .

We will now define the **application** of diffs Δ on theory graphs \mathcal{G} , which we denote by $\mathcal{G} \ll \Delta$. In MoC tools, this is sometimes called *patching*.

Definition 3. A diff Δ is called **applicable** to the theory graph \mathcal{G} if $\mathcal{G} \vdash \Delta$ according to the rules in Fig. 3.

$\frac{\mathcal{G} \vdash M \quad \mathcal{G} \not\vdash M?c}{\mathcal{G} \vdash \mathcal{A}(M, c : \omega = \omega')} \mathcal{A}_{dec}$	$\frac{\mathcal{G} \vdash M?c : \omega = \omega'}{\mathcal{G} \vdash \mathcal{D}(M, c : \omega = \omega')} \mathcal{D}_{dec}$		
$\frac{\vdash \mathcal{G}(T?c/o) = \omega}{\mathcal{G} \vdash \mathcal{U}(T, c, o, \omega, \omega')} \mathcal{U}_{sym}$	$\frac{\vdash \mathcal{G}(v?c/def) = \omega}{\mathcal{G} \vdash \mathcal{U}(v, c, def, \omega, \omega')} \mathcal{U}_{ass}$		
$\frac{\mathcal{G} \vdash T?c \quad \mathcal{G} \not\vdash T?c'}{\mathcal{G} \vdash \mathcal{R}(T, c, c')} \mathcal{R}_{dec}$			
<table style="width: 100%; border: none;"> <tr> <td style="width: 50%; border: none; text-align: center;"> $\frac{}{\mathcal{G} \vdash \cdot} \Delta_{base}$ </td> <td style="width: 50%; border: none; text-align: center;"> $\frac{\mathcal{G} \vdash \Delta \quad \mathcal{G} \ll \Delta \vdash \delta}{\mathcal{G} \vdash \Delta \bullet \delta} \Delta_{dec}$ </td> </tr> </table>		$\frac{}{\mathcal{G} \vdash \cdot} \Delta_{base}$	$\frac{\mathcal{G} \vdash \Delta \quad \mathcal{G} \ll \Delta \vdash \delta}{\mathcal{G} \vdash \Delta \bullet \delta} \Delta_{dec}$
$\frac{}{\mathcal{G} \vdash \cdot} \Delta_{base}$	$\frac{\mathcal{G} \vdash \Delta \quad \mathcal{G} \ll \Delta \vdash \delta}{\mathcal{G} \vdash \Delta \bullet \delta} \Delta_{dec}$		

Fig. 3. Applicability of Changes

Definition 4 (Change Application). Given a theory graph \mathcal{G} and a \mathcal{G} -applicable change δ , we define $\mathcal{G} \ll \delta$ as follows:

- If $\delta = \mathcal{A}(M, c : \omega = \omega')$ then $\mathcal{G} \ll \delta$ is the graph constructed from \mathcal{G} by adding the declaration $c : \omega = \omega'$ to module M .
- If $\delta = \mathcal{D}(M, c : \omega = \omega')$ then $\mathcal{G} \ll \delta$ is the graph constructed from \mathcal{G} by deleting the declaration $c : \omega = \omega'$ from module M .
- If $\delta = \mathcal{U}(M, c, o, \omega, \omega')$ then $\mathcal{G} \ll \delta$ is the graph constructed from \mathcal{G} by updating the component at $M?c/o$ from ω to ω' .
- If $\delta = \mathcal{R}(T, c, c')$ then $\mathcal{G} \ll \delta$ is the graph constructed from \mathcal{G} by renaming the declaration at $T?c$ to $T?c'$.

Moreover, we define $\mathcal{G} \ll \Delta$ by $\mathcal{G} \ll \cdot = \mathcal{G}$ and $\mathcal{G} \ll (\Delta \bullet \delta) = (\mathcal{G} \ll \Delta) \ll \delta$.

Running Example 2 (Continuing Ex. 1). We have $Rev_1 \ll \Delta = Rev_2$ where Δ is the diff: $\mathcal{D}(PL, bool : type = \perp) \bullet \mathcal{A}(PL, form : type = \perp) \bullet \mathcal{D}(PL, \vee : bool \rightarrow bool \rightarrow bool = \perp) \bullet \mathcal{A}(PL, \neg : form \rightarrow form = \perp)$. Alternatively, we could use a rename $\mathcal{R}(PL, bool, form)$ instead of the add-delete pair.

The following simple theorem permits lookups in a hypothetical patched theory graph. This is important for scalability in the typical case where a large \mathcal{G} should be neither changed nor copied:

Theorem 1. Assume a theory graph \mathcal{G} and a \mathcal{G} -applicable diff Δ . Then

$$\begin{aligned}
 \vdash (\mathcal{G} \ll \cdot)(M?c/o) &= \mathcal{G}(M?c/o) \\
 \vdash (\mathcal{G} \ll (\Delta \bullet \mathcal{A}(M, c : \omega = \omega')))(M?c/o) &= \begin{cases} \omega & \text{if } o = \text{tp} \\ \omega' & \text{if } o = \text{def} \end{cases} \\
 \vdash (\mathcal{G} \ll (\Delta \bullet \mathcal{D}(M, c : \omega = \omega')))(M?c/o) &= \text{undefined} \\
 \vdash (\mathcal{G} \ll (\Delta \bullet \mathcal{U}(M, c, o, \omega, \omega')))(M?c/o) &= \omega' \\
 \vdash (\mathcal{G} \ll (\Delta \bullet \mathcal{R}(M, c', c)))(M?c/o) &= (\mathcal{G} \ll \Delta)(M?c'/o) \\
 \vdash (\mathcal{G} \ll (\Delta \bullet _))(M?c/o) &= (\mathcal{G} \ll \Delta)(M?c/o)
 \end{aligned}$$

where $_$ is any change not covered by the previous cases.

Proof. This is straightforward to prove using the definitions.

We will now introduce and study an equivalence relation between diffs. Intuitively, two diffs are equivalent if their application has the same effect:

Definition 5. Given a theory graph \mathcal{G} , two \mathcal{G} -applicable diffs Δ and Δ' are called \mathcal{G} -equivalent iff $\mathcal{G} \ll \Delta = \mathcal{G} \ll \Delta'$. We write this as $\Delta \equiv^{\mathcal{G}} \Delta'$.

Our main theorem about change application is that diffs can be normalized. We need some auxiliary definitions first:

Definition 6. The *referenced URIs* of a change are defined as follows: For both $\mathcal{A}(M, c : \omega = \omega')$ and $\mathcal{D}(M, c : \omega = \omega')$ they are $M?c/\text{tp}$ and $M?c/\text{def}$, for $\mathcal{U}(M, c, o, \omega, \omega')$ it is only $M?c/o$, and for $\mathcal{R}(T, c, c')$ they are $T?c/\text{tp}$, $T?c/\text{def}$,

$T?c'/\text{tp}$ and $T?c'/\text{def}$. Two changes δ and δ' have a **clash** if they reference the same URI.

A diff Δ is called **minimal** if there are no clashes between any two changes in Δ . A minimal diff is called **normal** if it is of the form $\Delta_1 \bullet \Delta_2$ where Δ_1 contains no renames and Δ_2 contains only renames.

Theorem 2. *Reordering the changes in a minimal diff yields an equivalent diff.*

Proof. In a minimal diff, each change affects a different declaration so the order of application is irrelevant.

Definition 7. $\mathcal{G}' - \mathcal{G}$ is obtained as follows:

1. The diff Δ contains the following changes (in any order):

$$\begin{array}{lll} \mathcal{U}(M, c, o, \omega, \omega') & \text{for} & \mathcal{G}(M?c/o) = \omega, \quad \mathcal{G}'(M?c/o) = \omega', \quad \omega \neq \omega' \\ \mathcal{D}(M?c : \omega = \omega') & \text{for} & \mathcal{G} \vdash M?c : \omega = \omega', \quad \mathcal{G}' \not\vdash M?c \\ \mathcal{A}(M?c : \omega = \omega') & \text{for} & \mathcal{G}' \vdash M?c : \omega = \omega', \quad \mathcal{G} \not\vdash M?c \end{array}$$

2. We say that a pair (A, D) of changes in Δ matches if $A = \mathcal{A}(T, c : \omega = \omega')$ and $D = \mathcal{D}(T, c' : \omega = \omega')$. They match uniquely if there is no other A' that matches D and no other D' that matches A .
3. $\mathcal{G}' - \mathcal{G}$ arises from Δ by removing every uniquely matching pair (A, D) and appending the respective rename $\mathcal{R}(T, c, c')$.

This definition first generates an add or delete for every URI that exists only in \mathcal{G}' or \mathcal{G} , respectively, and 0 – 2 updates for every URI that exists in both. Then uniquely matching add-delete pairs are replaced with renames. The uniqueness constraint is necessary to make the last step deterministic.

Running Example 3 (Continuing Ex. 2). *The first step of the computation of the difference $\text{Rev}_2 - \text{Rev}_1$ yields the diff from Ex. 2. The next steps simplify this diff to $\mathcal{D}(PL, \vee : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool} = \perp) \bullet \mathcal{A}(PL, \neg : \text{form} \rightarrow \text{form} = \perp) \bullet \mathcal{R}(PL, \text{bool}, \text{form})$.*

Theorem 3. $\mathcal{G}' - \mathcal{G}$ is normal, \mathcal{G} -applicable, and $\mathcal{G} \ll (\mathcal{G}' - \mathcal{G}) = \mathcal{G}'$.

Proof. The proof is straightforward from the definition.

Theorem 4. *If $\mathcal{G}' = \mathcal{G} \ll \Delta$, then there is a normal diff Δ' such that $\Delta \equiv^{\mathcal{G}} \Delta'$.*

Proof. We put $\Delta' = (\mathcal{G} \ll \Delta) - \mathcal{G}$. Then the result follows from Thm. 3.

4.2 A Data Structure for Dependencies

As our validation units are the components of MMT declarations, we need to formulate the validity of MMT theory graphs in a way that permits separate validation of each component:

Definition 8. A theory graph \mathcal{G} is called **foundationally valid** if for all symbol or assignment declarations $\mathcal{G} \vdash M?c : \omega = \omega'$ (recall Not. [1](#)), we have $\mathcal{G} \vdash \omega' : \omega$.

Now we can make formal statements how the validity of a theory graph is affected by changes. First, a typical property of typing relations is that they satisfy a weakening property: Additional information can not invalidate a type inference:

Definition 9. A foundation is called **monotonous** if the following rules are admissible for any $A = \mathcal{A}(M, c : _ = _)$ and for any $U = \mathcal{U}(M, c, o, \perp, _)$:

$$\frac{\mathcal{G} \vdash \omega : \omega' \quad \mathcal{G} \vdash A}{\mathcal{G} \ll A \vdash \omega : \omega'} \qquad \frac{\mathcal{G} \vdash \omega : \omega' \quad \mathcal{G} \vdash U}{\mathcal{G} \ll U \vdash \omega : \omega'}$$

Almost all practical foundations for MMT are monotonous. This includes even substructural type theories like linear LF [CP02](#) because we only require weakening for the set of global declarations, not for local contexts. A simple counterexample is a type theory with induction in which constructors can be added as individual declarations: Then adding a constructor will break an existing induction. But most type theories introduce all constructors in the same declaration.

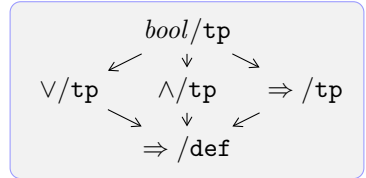
While monotony permits handling additions to a theory graphs in general, we must introduce dependency relations between components to handle updates and deletes. Intuitively, if a validation unit U does not depend on U' , then deleting U' is guaranteed not to affect the validity of U :

Definition 10. A **dependency relation** for a theory graph \mathcal{G} is a binary relation $\not\leftrightarrow$ between declaration components $M?c/o$ and $M'?c'/o'$ such that the following rules are admissible:

$$\frac{\mathcal{G} \vdash M?c/o = \omega'' \quad \mathcal{G} \vdash M'?c' : \omega = \omega' \quad M?c/o \not\leftrightarrow M'?c'/\text{tp}}{\mathcal{G} \ll \mathcal{U}(M, c, o, \omega'', \perp) \vdash \perp : \omega}$$

$$\frac{\mathcal{G} \vdash M?c/o = \omega'' \quad \mathcal{G} \vdash M'?c' : \omega = \omega' \quad M?c/o \not\leftrightarrow M'?c'/\text{def}}{\mathcal{G} \ll \mathcal{U}(M, c, o, \omega'', \perp) \vdash \omega' : \omega}$$

Note that dependency relations are not necessarily transitive. That way changes can be propagated one dependency step at a time, and intermediate revalidation can show that no further propagation is necessary. Of course, the transitive closure (in fact: any larger relation) is again a dependency relation. Our definition of a dependency relation was inspired by the one in [RKST1](#).



Running Example 4 (Continuing Ex. [3](#)). For the theory graph Rev_1 , we obtain a dependency relation by assuming a dependency whenever a constant occurs in a component. We also assume a dependency from each definiens to its type. The graph in the figure above illustrates this relation.

4.3 Change Propagation

It is tempting to study the propagation of only a change δ . But this does not cover the important case of transactions, where multiple changes are propagated together. This is typical in practice when an author makes multiple related changes. But it is very complicated to propagate an arbitrary diff. Our key insight is to focus on the **propagation of minimal diffs**. These are very easy to work with, and due to Thm. 3, this is not a loss of generality.

The central idea of our propagation algorithm is to introduce box terms that mark expressions as impacted. This has the advantage that propagation can be formalized as a closure operator on sets of changes so that no additional data structures for impacts are needed.

Definition 11. For a term ω and a rename $R = \mathcal{R}(T, c, c')$, we define ω^R as the term obtained from ω by replacing all occurrences of $T?c$ with $T?c'$. Similarly, if Δ contains only renames, we define ω^Δ by $\omega^\Delta \bullet R = (\omega^\Delta)^R$ and $\omega^\cdot = \omega$.

Definition 12. For the purposes of Def. 13, we say that a component $M?c/o$ is **modified** by Δ if Δ contains a change of the form $\mathcal{D}(M, c : _ = _)$ or $\mathcal{U}(M, c, o, \omega, _)$.

The following definition and theorem express our main result. We state them for the special case for a diff that does not add or delete assignments. The general case holds as well but is more complicated.

Definition 13 (Propagation). Assume a fixed theory graph \mathcal{G} and a fixed dependency relation \vartriangleright (which we omit from the notation). Assume a \mathcal{G} -applicable diff in normal form $\Delta = \Delta_1 \bullet \Delta_2$ that does not contain any adds or deletes of assignments. We define the propagation $\overline{\Delta}$ of Δ in multiple steps as follows:

1. Δ'_1 contains the following changes (in any order): whenever $M?c/o \vartriangleright M'?c'/o'$ and $M?c/o$ is modified by Δ_1 , the change

$$\mathcal{U}(M', c', o', \omega, \boxed{\omega}) \quad \text{for} \quad \mathcal{G} \ll \Delta \vdash M'?c'/o' = \omega$$

2. Δ'_2 contains the following changes (in any order): whenever $\mathcal{R}(T, c, _)$ $\in \Delta_2$ and $T?c/o \vartriangleright M'?c'/o'$, the change

$$\mathcal{U}(M', c', o', \omega, \omega^{\Delta_2}) \quad \text{for} \quad \mathcal{G} \ll \Delta \bullet \Delta'_1 \vdash M'?c'/o' = \omega$$

3. Δ'_3 contains the following changes for every morphism $\mathcal{G} \vdash v : T \rightarrow T'$ (in any order):

– whenever $\mathcal{A}(T, c : _ = _)$ $\in \Delta$ or $\mathcal{U}(T, c, \text{def}, _, _)$ $\in \Delta$, the change

$$\mathcal{A}(v, c := \boxed{\cdot})$$

– whenever $\mathcal{D}(T, c : _ = _)$ $\in \Delta$ or $\mathcal{U}(T, c, \text{def}, _, _)$ $\in \Delta$, the change

$$\mathcal{D}(v, \text{Ass}) \quad \text{for} \quad \mathcal{G} \ll \Delta \bullet \Delta'_1 \bullet \Delta'_2 \vdash v?c = \text{Ass}$$

- whenever $\mathcal{R}(T, c, c') \in \Delta$ and $\mathcal{G} \ll \Delta \bullet \Delta'_1 \bullet \Delta'_2 \vdash T?c/\text{def} = \perp$ and $\mathcal{G} \ll \Delta \bullet \Delta'_1 \bullet \Delta'_2 \vdash v?c := \omega$, the changes

$$\mathcal{D}(v, c := \omega), \mathcal{A}(v, c' := \omega)$$

4. $\overline{\Delta}$ is obtained as $\Delta'_1 \bullet \Delta'_2 \bullet \Delta'_3$.

Intuitively, Δ'_1 updates all impacted terms to box terms. If $\varphi \rightarrow$ is transitive, this includes all terms that depended on the now boxed terms. Δ'_2 updates all references to renamed declarations to the new name.

Δ'_3 ensures that all morphisms have exactly one assignment for every undefined constant in the domain. The first two subcases add empty assignments or delete existing ones if necessary. The third subcase renames those assignments where the corresponding constant in the domain has been renamed.

Theorem 5. *Consider the situation of Def. 13. Assume that the foundation is monotonous, that \mathcal{G} is foundationally valid, and that $\varphi \rightarrow$ is transitive. Let $\mathcal{G}' = \mathcal{G} \ll \Delta \bullet \overline{\Delta}$, and let \mathcal{G}^* be a theory graph that arises from \mathcal{G}' by replacing every box term with a term that type checks in the sense of Def. 8. Then \mathcal{G}^* is foundationally valid.*

Proof. Let us first consider the special case without renames or morphisms. We apply Def. 8 to \mathcal{G}^* . Due to Δ'_1 and the transitivity of $\varphi \rightarrow$, all possibly ill-typed terms have been replaced with box terms in \mathcal{G}' ; and according to the assumptions, these are replaced with well-typed terms in \mathcal{G}^* . Thus, the claim follows.

If there are renames, care must be taken to update all references to the renamed declarations. If there are adds, care must be taken to guarantee the totality of morphisms. Both conditions are already fulfilled in \mathcal{G}' . We omit the details.

A typical situation where we would apply Thm. 5 is after a user made the changes Δ . Then propagation marks all terms that have to be revalidated and — if not well-typed — replaced interactively with well-typed terms. The theorem guarantees the resulting graph is valid again.

$$\begin{aligned}
 PL = \{ & \\
 & \text{form : type} = \perp \\
 & \neg : \text{form} \rightarrow \text{form} = \perp \\
 & \wedge : \text{form} \rightarrow \text{form} \rightarrow \text{form} = \perp \\
 & \Rightarrow : \text{form} \rightarrow \text{form} \rightarrow \text{form} \\
 & \quad = \boxed{\lambda x. \lambda y. y \vee (x \wedge y)} \\
 & \}
 \end{aligned}$$

Running Example 5 (Continuing Ex. 3 and 4). *Using $\Delta = \text{Rev}_2 - \text{Rev}_1$ and the dependency relation from Ex. 4, we compute $\overline{\Delta}$. First $\Delta = \Delta_1 \bullet \Delta_2$ with $\Delta_1 = \mathcal{D}(PL, \vee : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool} = \perp) \bullet \mathcal{A}(PL, \neg : \text{form} \rightarrow \text{form} = \perp)$ and $\Delta_2 = \mathcal{R}(PL, \text{bool}, \text{form})$. Then*

$$\overline{\Delta}_1 = \mathcal{U}(PL, \Rightarrow, \text{def}, \lambda x. \lambda y. y \vee (x \wedge y), \boxed{\lambda x. \lambda y. y \vee (x \wedge y)})$$

as well as $\overline{\Delta}_2 = \mathcal{U}(PL, \wedge, \text{tp}, \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}, \text{form} \rightarrow \text{form} \rightarrow \text{form}) \bullet \mathcal{U}(PL, \Rightarrow, \text{tp}, \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}, \text{form} \rightarrow \text{form} \rightarrow \text{form})$ and $\overline{\Delta}_3 = \cdot$. Finally, the theory graph $\text{Rev}_1 \ll \Delta \bullet \overline{\Delta}$ is shown above. As stated in Thm. 5, it becomes foundationally valid after replacing the box term with a term of the right type.

5 A Generic Change Management API

Implementation We have implemented the data structures and algorithms from Sect. 4 as a part of the MMT API [Rab08]. In fact, our implementation covers a much larger fragment of MMT than discussed in this paper.

In particular, the API now contains functions that compute the **difference** $\mathcal{G}' - \mathcal{G}$ of two theory graphs. The difference of two modules can be computed as well. The two arguments can either be provided directly or the previous revision can be pulled automatically from an SVN repository.

We also added functions for **change propagation** that enrich a normal diff with its direct impacts according to Def. 13. The generated box terms are represented as OPENMATH error objects. During the propagation algorithm, we make crucial use of Thm. 1 to increase the efficiency.

Both of these algorithms are implemented foundation-independently. The foundation is only needed to obtain the dependency relation and to revalidate the impacted declarations. Both are special cases of type checking.

The MMT API relegates a **type checking** obligation $\omega' : \omega$ to a plugin for the respective foundation. In particular, there is a plugin for a monotonous foundation for the logical framework LF [HHP93], which induces implementations of type checking for all formal systems represented in LF (i.e., for a lot of formal systems [CHK+11]).

The plugin interface is such that the plugin calls back to the main system whenever it needs to look up any component $M?c/o$. In the simplest case, we can trace these callbacks to obtain the set of components $Used(\omega', \omega)$ that were used to validate $\omega' : \omega$. When the system validates a theory graph \mathcal{G} according to Def. 8, we obtain a **dependency relation** by putting for every symbol or assignment $\mathcal{G} \vdash M'?c' : \omega = \omega'$

$$\begin{aligned} M?c/o \looparrowright M'?c'/\text{tp} & \quad \text{if} \quad M?c/o \in Used(\perp, \omega) \\ M?c/o \looparrowright M'?c'/\text{def} & \quad \text{if} \quad M?c/o \in Used(\omega', \omega) \\ M'?c'/\text{tp} \looparrowright M'?c'/\text{def} & \end{aligned}$$

Note that we first check the type of the declaration and then separately check the definiens against that type even though the latter implies the former. This is important because the type will usually have much less dependencies than the definiens.

This dependency relation is stored in the MMT ontology, which MMT maintains together with the content [HIJ+11]. Alternatively, the foundation can explicitly provide a dependency relation, or we can import dependency relations externally, e.g., the ones from [AMU11].

Application We have applied the resulting system to obtain a change management API for the LATIN library. Using the MMT plugins for LF — the language underlying the LATIN library — we obtain a foundation that validates the library and computes a dependency relation for it. Fig. 4 gives a summary of the

dependency relation, where we include only the about 1700 components falling into the fragment of MMT treated in this paper. The tables group the components by the number of components that they depend on (left) or that depend on them (right). This includes only direct dependencies — taking the transitive closure increases the numbers by about 20 %.

dependencies	components (%)	impacts	components (%)
0 – 5	1373 (79)	0 – 5	1504 (86.5)
6 – 10	271 (15.6)	6 – 10	101 (5.8)
11 – 15	81 (4.7)	11 – 25	76 (4.4)
16 – 26	13 (0.7)	26 – 50	31 (1.8)
		50 – 449	26 (1.5)

Fig. 4. Components grouped by dependencies and impacts

The number of dependencies and impacts is generally low. This is a major benefit of our choice of using type and definiens as separate validation units, which avoids the exponential blowup one would otherwise expect. Indeed, on average a type has 3 times as many impacts as a definiens.

Our differencing algorithm can detect and propagate changes easily, and it is straightforward to revalidate the impacted components. The numbers show that even manual inspection (as opposed to automatic revalidation) is feasible in most cases: For example, changes to 86 % of the components impact only 5 or less components. Even if the number of impacted components is so small, it is usually very difficult for humans to identify exactly which components are impacted. Our MoC infrastructure, on the other hand, does not only identify them automatically but also guarantees that all other components stay valid.

6 Conclusion

We have presented a theory of change management based on the MMT language including difference, dependency, and impact analysis. As MMT is foundation-independent, our work yields a theory of change management for an arbitrary declarative language. Our work is implemented as a part of the MMT API and thus immediately applicable to any language that is represented in MMT. The latter includes in particular the logical framework LF and thus every language represented in it.

Because we use fine-grained dependencies, change propagation can identify individual type checking obligations (which subsume proof obligations) that have to be revalidated. The MMT API already provides a scalable framework for validating individual such obligations efficiently. Therefore, our work provides the foundation for a large scale change management system for declarative languages.

While our presentation has focused on a small fragment of MMT, the results can be generalized to the whole MMT language, in particular the module system.

Presently the most important missing feature is a connection between the MMT abstract syntax and the concrete syntax of individual languages. Therefore, change management currently requires an export into MMT's abstract syntax (which exists for, e.g., Mizar [TB85], TPTP [SS98], and OWL [W3C09]). Consequently, **future work** will focus on developing fast bidirectional translations between human-friendly source languages and their MMT content representation. If these include fine-grained cross-references between source and content, MMT can propagate changes into the source language; this could happen even while the user is typing.

More generally, this approach extends to pure mathematics where the source language is, e.g., \LaTeX . If the source is formalized manually, it is sufficient to include cross-references in the above sense. Then changes in the \LaTeX source can be treated and propagated like changes in the formalization. Alternatively, we can avoid a manual formalization if certain annotations are present in the source: firstly, annotations that map a line number to the identifier of the statement (definition, theorem, etc.) made at that line; secondly, annotations that explicate the dependency relation between statements. For example, the \LaTeX package for \LaTeX permits such annotations in a way that supports automated extraction.

References

- ADD⁺11. Autexier, S., David, C., Dietrich, D., Kohlhase, M., Zholudev, V.: Workflows for the Management of Change in Science, Technologies, Engineering and Mathematics. In: Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.) MKM 2011 and Calculemus 2011. LNCS (LNAI), vol. 6824, pp. 164–179. Springer, Heidelberg (2011)
- AHM10. Autexier, S., Hutter, D., Mossakowski, T.: Change Management for Heterogeneous Development Graphs. In: Siegler, S., Wasser, N. (eds.) Walther Festschrift. LNCS, vol. 6463, pp. 54–80. Springer, Heidelberg (2010)
- AHMS99. Autexier, S., Hutter, D., Mantel, H., Schairer, A.: Towards an Evolutionary Formal Software-Development Using CASL. In: Bert, D., Choppy, C., Mosses, P.D. (eds.) WADT 1999. LNCS, vol. 1827, pp. 73–88. Springer, Heidelberg (2000)
- AHMS02. Autexier, S., Hutter, D., Mossakowski, T., Schairer, A.: The Development Graph Manager MAYA. In: Kirchner, H., Ringeissen, C. (eds.) AMAST 2002. LNCS, vol. 2422, pp. 495–502. Springer, Heidelberg (2002)
- AM10. Autexier, S., Müller, N.: Semantics-Based Change Impact Analysis for Heterogeneous Collections of Documents. In: Gormish, M., Ingold, R. (eds.) Proceedings of 10th ACM Symposium on Document Engineering (DocEng2010) (2010)
- AMU11. Alama, J., Mamane, L., Urban, J.: Dependencies in Formal Mathematics. CoRR, abs/1109.3687 (2011)
- Apa00. Apache Software Foundation. Apache Subversion (2000), <http://subversion.apache.org/>
- BC08. Bundy, A., Chan, M.: Towards Ontology Evolution in Physics. In: Hodges, W., de Queiroz, R. (eds.) WoLLIC 2008. LNCS (LNAI), vol. 5110, pp. 98–110. Springer, Heidelberg (2008)

- BCC⁺04. Buswell, S., Caprotti, O., Carlisle, D., Dewar, M., Gaetano, M., Kohlhasse, M.: The Open Math Standard, Version 2.0. Technical report, The Open Math Society (2004), <http://www.openmath.org/standard/om20>
- CHK⁺11. Codescu, M., Horozal, F., Kohlhasse, M., Mossakowski, T., Rabe, F.: Project Abstract: Logic Atlas and Integrator (LATIN). In: Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.) MKM 2011 and Calculemus 2011. LNCS, vol. 6824, pp. 289–291. Springer, Heidelberg (2011)
- CP02. Cervesato, I., Pfenning, F.: A Linear Logical Framework. *Information and Computation* 179(1), 19–75 (2002)
- CVS. Concurrent Versions System: The open standard for Version Control, Web site at <http://cvs.nongnu.org/> (seen February 2012)
- EG89. Ellis, C., Gibbs, S.: Concurrency control in groupware systems. In: Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, pp. 399–407. ACM (1989)
- FGT92. Farmer, W., Guttman, J., Thayer, F.: Little Theories. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 467–581. Springer, Heidelberg (1992)
- Git. Git, Web Site at <http://git-scm.com/>
- HHP93. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. *Journal of the Association for Computing Machinery* 40(1), 143–184 (1993)
- HIJ⁺11. Horozal, F., Iacob, A., Jucovschi, C., Kohlhasse, M., Rabe, F.: Combining Source, Content, Presentation, Narration, and Relational Representation. In: Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.) MKM 2011 and Calculemus 2011. LNCS, vol. 6824, pp. 212–227. Springer, Heidelberg (2011)
- Hut00. Hutter, D.: Management of change in structured verification. In: Proceedings Automated Software Engineering, ASE 2000, pp. 23–34 (2000)
- KK11. Kohlhasse, A., Kohlhasse, M.: Versioned links. In: Proceedings of the 29th Annual ACM International Conference on Design of Communication (SIGDOC), (2011)
- Rab08. Rabe, F.: The MMT System (2008), <https://trac.kwarc.info/MMT/>
- RK11. Rabe, F., Kohlhasse, M.: A Scalable Module System (2011), <http://arxiv.org/abs/1105.0548>
- RKS11. Rabe, F., Kohlhasse, M., Sacerdoti Coen, C.: A Foundational View on Integration Problems. In: Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.) MKM 2011 and Calculemus 2011. LNCS, vol. 6824, pp. 107–122. Springer, Heidelberg (2011)
- SS98. Sutcliffe, G., Suttner, C.: The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning* 21(2), 177–203 (1998)
- TB85. Trybulec, A., Blair, H.: Computer Assisted Reasoning with MIZAR. In: Joshi, A. (ed.) Proceedings of the 9th International Joint Conference on Artificial Intelligence, pp. 26–28 (1985)
- W3C09. W3C. OWL 2 Web Ontology Language (2009), <http://www.w3.org/TR/owl-overview/>
- Wag10. Wagner, M.: A change-oriented architecture for mathematical authoring assistance. PhD thesis, Universität des Saarlands (2010)

MathWebSearch 0.5: Scaling an Open Formula Search Engine

Michael Kohlhase, Bogdan A. Matican, and Corneliu-Claudiu Prodescu

Computer Science, Jacobs University Bremen
<http://kwarc.info>

Abstract. MATHWEBSEARCH is an open-source, open-format, content-oriented search engine for mathematical formulae. It is a complete system capable of crawling, indexing, and querying expressions based on their functional structure (operator tree) rather than their presentation.

In version 0.5, we concentrate on scalability issues in MATHWEBSEARCH to take advantage of corpora in the giga-formula range. We re-implemented the index to make it distributable and made all the APIs web standards conformant. Our experiments show that this architecture results in a scalable application.

1 Introduction

As the world of information technology grows, being able to quickly search data of interest becomes one of the most important tasks in any kind of environment, be it academic or not. Here we tackle the problem of finding information that is given in the form of (mathematical) formulae. Standard search engines like GOOGLE cannot deal with formulae at all, severely limiting the reach and utilization of technical, scientific, and engineering documents.

In this paper we present new work in the context of the MATHWEBSEARCH system; a search engine that addresses the problem of searching mathematical formulae from a semantic point of view, it finds formulae by their structure and meaning not via their presentation.

In [KS06] we have presented the motivation, query language, and web front end of MATHWEBSEARCH 0.1. In [KK07] we have re-examined the value proposition of semantic search for mathematical knowledge homing in on the benefits and sacrifices induced by the various search approaches [You06b, MM06, LM06], from a user's perspective. The result of this analysis is MATHWEBSEARCH 0.5, which we describe in this paper. The new version features significant efficiency gains (space efficiency increased by a factor of five), new management features, advanced searching capabilities, and a new user interface. The MATHWEBSEARCH system (see [MWS] for details) is released under the Gnu General Public License.

The motivation for the work reported in this paper is the availability of large corpora, such as the arXMLiv corpus [SK08] with almost three quarters of a million scientific articles and an estimated giga-formula. This has not only re-kindled

¹ We deem a corpus as *large* if it has more than 20 million expressions

interest in formula search², but also severely taxes the scalability of systems. Scalability issues for presentation-based search engines have been addressed in [SL11]. Such engines map formulae to “special words” which can then be indexed by conventional bag-of-word information retrieval engines, which have become extremely scalable over the last years. The case for MATHWEBSEARCH is completely different, since the content-based unification queries it offers require an index data structure that reflects the inner structure of formulae (rather than just pointers to words). Even with the space efficiency gains in MATHWEBSEARCH 0.5, the indices will surpass the main memory of most machines. Therefore, we have laid the foundations for distributing the MATHWEBSEARCH in this version.

Before we present MATHWEBSEARCH 0.5 from a technical perspective in Section 3, we will recap unification-based querying. We evaluate the system on a large corpus in Section 4 and see that we need to distribute MATHWEBSEARCH to cope with linear RAM usage. Section 5 presents the necessary extensions of the indexing. Section 6 concludes the paper and discusses future work.

2 Querying Mathematics by Unification

Retrieval of mathematical knowledge and information via unification-based queries for content-encoded mathematical formulae is very natural. In [KS06] we have already discussed **instantiation queries**, which can be used to retrieve partially remembered formulae, e.g. the query for the formula for energy of a given signal $s(t)$ in Figure 1. Note that instantiation queries are more expressive as a query language than e.g. regular expressions supported by some text-based search engine, since we can use variable co-occurrences to query for co-occurring subterms.

Query (query variables marked as named boxes)	Result (Parseval's Theorem)
$\int_{\min}^{\max} \boxed{f}(x)^2 dx$	$\frac{1}{T} \int_0^T s^2(t) dt = \sum_{k=-\infty}^{\infty} \ c_k\ ^2$

Fig. 1. An Instantiation Query

To see the full power of unification-based querying consider a student who encounters $\int_{\mathbb{R}^2} |\sin(t) \cos(t)| dt$ and wishes to know if there are any mathematical statements (like theorems, identities, inequalities) that can be applied to it. Indeed, there are many such statements (for example Hölder's inequality) and they can be found using **generalization queries**. The idea behind answering generalization queries is that the index marks universal³ variables in subterms

² The next NTCIR-10 Challenge in spring 2013 will have a “math track”. NTCIR evaluates information access technologies in a series of competition events in Japan.

³ We consider an identifier as universal if it can be instantiated without changing the truth value of the containing expression. In formal representations like first-order logic, such variable occurrences can be effectively computed, but in semi-formal settings like mathematical textbooks, they have to be approximated by heuristic methods; see the discussion in the conclusion for details.

as generalization targets. Hence, the search engine looks for terms in the index which, after instantiating the universal identifiers, become equal to the query. For our example, we have in the index the term (we reuse the box notation for generalization targets in the index) in Figure 2, which the search engine instantiates $\boxed{x} \mapsto t, \boxed{f} \mapsto \sin, \boxed{g} \mapsto \cos, \boxed{D} \mapsto \mathbb{R}^2$ in order to find the generalization query. Note that the variant query $\int_{\mathbb{R}^2} |\sin(t) \cos(2t)| dt$ will not find Hölder’s inequality since that would introduce inconsistent substitutions $\boxed{x} \mapsto t$ and $\boxed{x} \mapsto 2t$.

$$\int_{\boxed{D}} |\boxed{f}(\boxed{x}) \boxed{g}(\boxed{x})| d\boxed{x} \leq \left(\int_{\boxed{D}} |\boxed{f}(\boxed{x})|^{\boxed{p}} d\boxed{x} \right)^{\frac{1}{\boxed{p}}} \left(\int_{\boxed{D}} |\boxed{g}(\boxed{x})|^{\boxed{q}} d\boxed{x} \right)^{\frac{1}{\boxed{q}}}$$

Fig. 2. A Formula with Universal Variables in the Index

A very similar idea is used in **variation queries** where the indexed terms are searched to match the search expression but only renamings of generic terms are allowed. This type of queries prove to be helpful when the structure of the term needs to be maintained.

Sometimes, however, one is in the position that the searching criteria is somewhere between instantiation queries (i.e. parts are unknown) and generalization queries (parts are probably instantiated already). In this case we give the possibility to pose **unification queries**. As the name suggests, the query just finds terms which are unifiable with the search expression. A query like $g^2 \cos(\boxed{x}) + b \sin(\sqrt{\boxed{y}})$ would match the term $\boxed{a} \cos(\boxed{t}) + \boxed{b} \sin(\boxed{t})$ as we can substitute $\boxed{x} \mapsto \sqrt{\boxed{y}}, \boxed{t} \mapsto \sqrt{\boxed{y}}, \boxed{a} \mapsto g^2, \boxed{b} \mapsto b$ to get the term $g^2 \cos(\sqrt{\boxed{y}}) + b \sin(\sqrt{\boxed{y}})$.

3 The MathWebSearch System, Version 0.5

The MATHWEBSEARCH system consists of the three main components pictured in Figure 3. The *crawler subsystem* collects data from the corpora⁴. It transforms the mathematical formulae in the corpus into *MWS Harvests* (XML files that contain formula-URIreference pairs) and feeds them into the core system. The *core system* (the MATHWEBSEARCH daemon mwsd) builds the search index and processes search queries: it accepts the MATHWEBSEARCH input formats (*MWS Harvest* and *MWS Query*; see [KP]) and generates the MATHWEBSEARCH output format (*MWS Answer Set*). These are communicated through the *RESTful interface* restd which provides a public HTTP API conforming to the REST paradigm.

⁴ Note that we envision essentially one crawler per corpus. The crawlers are specialized to the respective formula representation, the organization and access methods to the corpus, etc. We have only implemented a crawler for the ARXIV (see Section 4), but additional crawlers can be patterned after this (see Section 6.1).

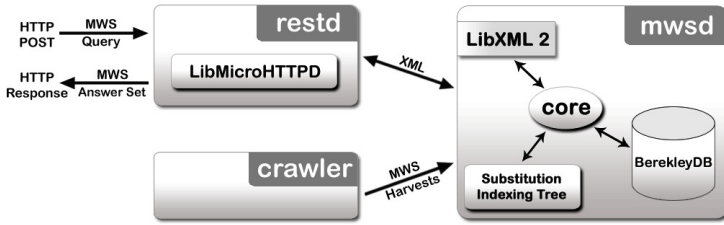


Fig. 3. MWS-0.5 System Structure

These components have been implemented using POSIX-compliant [\[Pos\]](#) C++. We use the MicroHTTPd library [\[Mic\]](#) API for handling HTTP, and LibXML2 [\[Vei\]](#) API for XML parsing. The meta-data accompanying the internal index is stored using an external database system. As we are dealing mainly with key-value retrieval, the BerkeleyDB [\[Ber\]](#) API was preferred.

The system supports two main workflows:

1. The crawler sends an *MWS Harvest* to mwsd. The XML is parsed and an internal representation is generated. This is used to update the Substitution Indexing Tree and consequently the database.
2. The user sends an *MWS Query* to mwsd. The XML is parsed, an internal query is generated. Using an efficient traversal of the index tree, formulas matching the search term are retrieved and aggregated into a result. This is translated to an *MWS Answer Set* and sent back to the user.

3.1 Substitution Tree Indexing in MathWebSearch

As we are interested in indexing mathematical formulae at a large scale (document archives, text corpora), repetitive content is expected. After all, theorems are built on top of other theorems and terms on top of subterms. With this in mind, we chose a space-efficient internal representation based on substitutions.

In the previous version of MATHWEBSERCH, we used a technique borrowed from Automated Theorem Proving called Substitution Indexing [\[Gra96\]](#). It involves indexing expressions in a tree based on generality. The root is a generic variable and, as we go from a node to one of its children, one or more substitutions occur. For this version, we kept the substitution tree model and performed a few changes to better fit our design goals.

As such, we improved query times, by imposing a fixed substitution ordering. Hence, the query term describes a deterministic path through the index. The chosen ordering instantiates the left-most variable

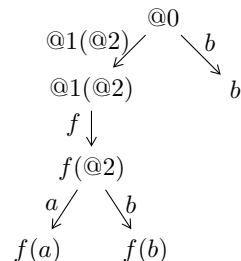


Fig. 4. Example DFS Substitution Tree

first, equivalent to DFS traversal of the operator tree. An example index, containing the terms $f(a)$, $f(b)$ and b , is presented in Figure 4. Note that the edges in the tree are labeled with the operators and operations: @1(@2) stands for the application operation.

Additionally, we save space by performing two steps of pre-processing for inserted terms, as well as query terms.

Firstly, we detect and reduce identical subterms. This is done by breaking the term into all possible subterms and detect equivalent subterms. Following this, the term is rewritten using only unique subterms (repeated matches are replaced through references to the first match in DFS order). For example the term $f(a, g(b), g(b))$ can be rewritten to $f(a, g(b), @[4])$, where @[4] represents the 4th term in DFS order: the subterms in DFS order are: $f(a, g(b), @[4])$, f , a , $g(b)$, g , b .

Furthermore, query terms with repeating identical query variables are reduced and handled as query terms with no repeating query variables. More importantly, this makes the search process stateless, as no previously matched query variable instantiations need to be stored.

Secondly, we hold an internal dictionary which maps symbols (in CONTENT-MATHML, represented by element name, attributes and text content) to integer IDs. The encoding relies on the fact that there are relatively few distinct tokens (compared to the number of expressions, for example). This achieves significant memory savings at a small price, since each (inserted or query) term is encoded exactly once.

3.2 Search Front Ends and Embeddings

For practical applications, mwsd serves as a search back-end that needs to be embedded into a front-end system, which hides some of the complexities of writing MATHWEBSEARCH queries from the user. One example of a front-end system we are experimenting with is given in Figure 5. Here the user can enter queries in the \LaTeX extended with the `\qvar` macro for query variables. This is then transformed into the content-MathML-based MATHWEBSEARCH queries by the \LaTeX XML daemon [GSK11] (the formula is also presented to the user with the query variables colored red). Upon receiving the resulting URIs, the front-end assembles a list of formulae and paper titles which link to the original paper. In this situation we are making use of the fact that \TeX/\LaTeX is a lingua franca for technical communication in the Mathematics community. For other communities, leveraging the MS Office equation editors might be an attractive option. For active document settings (e.g. in semantic publishing systems like Planetary [Koh+11]), formulae might be instrumented with a “search similar formulae” interaction. The same holds for integrated semantic development system such as Mathematica.

Questions
Activity
Sign In
Books
Articles
MWS Engine BETA

```

lim y
x→0
<m:apply>
<m:apply>
  <m:csymbol
cd="ambiguous">subscript</m:csymbol>
<m:limit/>
<m:apply>
<m:ci>→</m:ci>

```

$$\chi(t, t_w) = \lim_{h_0 \rightarrow 0} \frac{m[h](t)}{h_0}$$

$$\lim_{\mu, \mu_0 \rightarrow 0} I_1^c(\mu, \mu_0, \phi - \phi_0) = \frac{aF_0}{4(c+1)}$$

$$\lim_{\mu, \mu_0 \rightarrow 0} I_1^c(\mu, \mu_0, \phi - \phi_0)$$

1 2 next

Generalized off-equilibrium fluctuation-dissipation relations in random Ising systems

Author: Federico Ricci-Tersenghi
<ricci@chimera.roma1.infn.it>

Behavior of the reflection function of a plane-parallel medium for directions of incidence and reflection tending to horizontal directions

Author: Daphne Stam <d.m.stam@sron.nl>

Behavior of the reflection function of a plane-parallel medium for directions of incidence and reflection tending to horizontal directions

Author: Daphne Stam <d.m.stam@sron.nl>

Fig. 5. MWS-0.5 arXivDemo Search Interface

4 Evaluation

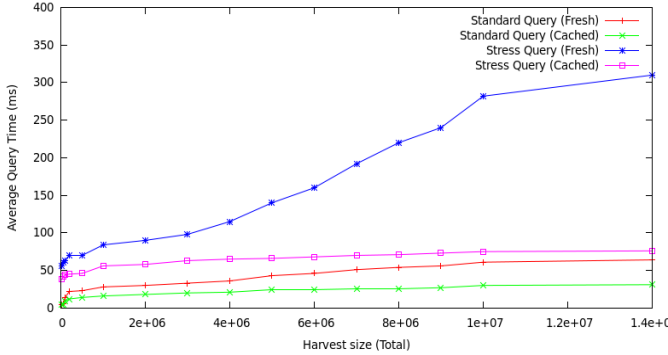
We evaluate the MATHWEBSEARCH implementation on a large corpus of mathematical formulae:

The arXMLiv Corpus. Our group is working on the translation of the almost 750.000 T_EX/L^AT_EX articles on Physics, Mathematics, and Computer Science in the Cornell ePrint archive (see <http://www.arXiv.org>) to MATHML [SK08]. **The arXMLiv corpus** is the result of translating ~72% of the arXiv papers. For our evaluation we have harvested ca 65% (the fragment that have been converted without errors), resulting in a total of 115 million expressions. A trivial estimation suggests that the full arXMLiv corpus would contain approx. 245 million formulae. To harvest these, the ARXMLIV crawler goes recursively through the pages of [arXMLiv](#) extracting the content MATHML⁵ elements, combines them with URI references, and reports them to MATHWEBSEARCH. We will now report on a performance analysis for MATHWEBSEARCH parametrized on **harvest size** (see Figure 6). As our index also indexes subformulae, we include them in the harvest size. Note that in the arXMLiv corpus a formula has 5.6 proper subformulae on average⁶ so we estimate the number of indexable formula

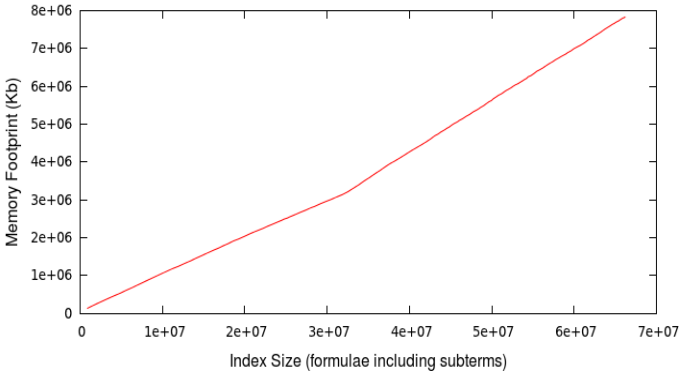
⁵ The result of the transformation contains both Content and Presentation MATHML representations in parallel markup.

⁶ This rather low number comes from the fact that roughly 2/3 of the formulae in the arXMLiv corpus consist of only one letter; these are largely irrelevant for search purposes.

occurrences in the arXiv corpus to be $6.6 \times 245 \times 10^8$ or 1.6 billion. Note that many of these formulae will actually be identical, leading to space savings in the index: recall that the URIs of the subformula occurrences are stored in a database indexed by the leaves of the index (see Figure 3). Thus the index grows with the formula, whereas the database grows with the formula occurrences.



(a) Query Times



(b) Memory Usage

Fig. 6. Experimental Performance Analysis

Average query times. We'll start with the time efficiency aspect, as this is highly relevant for a search system. The average query times [\[7\]](#) are presented in Figure [6\(a\)](#). This experiment involves measuring several query response times. A standard query has anszsize = 30 and limitmin < 9000. Response times are measured for standard queries (fresh and cached) from 0 up to 5 qvars. Additionally, stress queries with anszsize = 10000 are used.

As one can see, the query response times are fairly constant as the harvest data increases. This fits with the theory, as the querying process will follow

⁷ Note that the queries were sent from the local network, to eliminate any channel delays.

the same paths in the index (because the query data remains constant). The small increase is due to the slightly higher density of the index tree (which affects retrieval of the right path). The gap between the fresh and cached stress queries is expected, due to the fact that the current bottleneck is retrieving the meta-data from the external database. Hence, caching significantly improves the results. In comparison to MATHWEBSEARCH 0.1 [KS06], which reported query times below 50 ms for simple queries and 200 ms for stress queries on a harvest size of 1.6 million, we see that the query times have not increased (even after normalization for hardware effects).

Memory usage. The graph in Figure 6(b) presents the memory footprint of the mwsd process, as the system indexed 11.5 million expressions (67 million including subterms) harvested from the arXMLiv corpus. In comparison to MATHWEBSEARCH 0.1, which reported a memory footprint of 770 MB for a harvest size of 1.6 million, we see a space efficiency improvement by a factor of five. Contrary to our expectations of logarithmic increase, we see a fairly linear graph; the fact that the gradient became steeper after ~ 33 million expressions is particularly unexpected. We are still investigating this.

Nevertheless, the experimental results are valuable to estimate the memory necessary to index the entire arXiv corpus. Assuming linear scaling across the 245 million formulae estimated for the arXiv, the memory necessary to index all the formulae would be $245 \times 8/11.5 \approx 170Gb$ according to our experiment. As this transcends the RAM of most machines, we have extended mwsd so that it can be distributed: a reasonable size computer cluster could easily accommodate the entire arXMLiv corpus and thus provide content-based formula search for arXiv.org.

5 Distributing MathWebSearch

We are currently implementing a distributed version of MATHWEBSEARCH. The core components, like the RESTful interface, the data formats, and the indexing data structures, remain the main building blocks. We complement them with a distributable version of our storage data structure, data persistency, migration and load balancing. Next, we will present some of the design decisions.

5.1 A Distributable Substitution Tree

As presented in Section 3, the main indexing data structure is a DFS substitution tree. To represent the tree in a manner that supports cross-machine links, we use three types of index nodes:

Internal Index Nodes are used to navigate through most parts of the tree.

Their data stores mappings from token ID to the corresponding index node.

Leaf Index Nodes represent the end of a particular formula and its corresponding ID in the URL+URI database.

Remote Index Nodes represent cross-sector links. Their data consists of a pair of memory sector ID and node ID, which uniquely determine the corresponding index node.

As harvests are fed into the system, the index is built. Instead of building it on the heap (with no control over individual node’s memory locations), the system places index nodes inside specific *memory sectors*.

Memory sectors are basically the smallest units of replication, migration, and load balancing. Each consist of a forest of index node trees packed into a continuous area of memory of fixed size⁸. Also, for each memory sector, we have an attached *local database* corresponding to the leaf index node results. As the index grows, individual sectors get filled with these trees. When a sector reaches a given threshold, two new sectors are created and the contents of the original sector get split between the new ones.

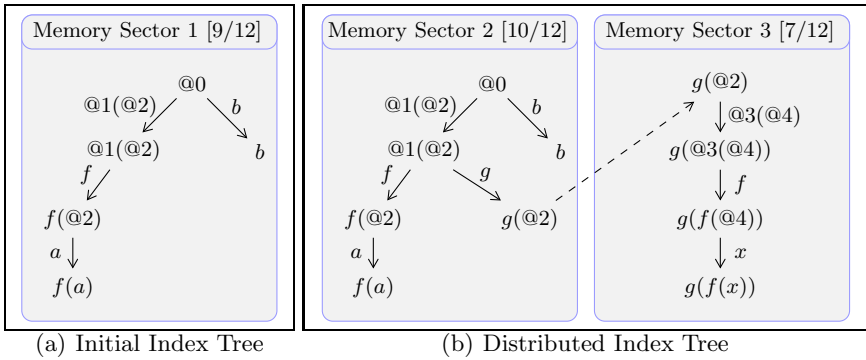


Fig. 7. Tree Distribution across Memory Sectors

In Figure 7, we present an example of term insertion which causes a tree split. Consider the initial tree containing the expressions $f(a)$ and b and memory sectors of size 12 units, as depicted in Figure 7(a). We consider a memory model where each link and each node costs 1 unit. Let’s insert $g(f(x))$. As the resulting tree would overflow the current memory sector, a new one is created and parts of the tree are migrated to this new sector (See Figure 7(b)). To split the tree, the system performs a DFS traversal. As it traverses, the algorithm monitors two factors:

- the size of the already explored part of the tree (equivalent to the proportion of the split)
- the DFS fringe (each internal index node in here will generate remote index nodes).

⁸ The exact size depends on the RAM sizes of the nodes where the system will run, typical values ranging between 256Mb and 2048Mb.

We want to minimize the number of remote index nodes, while making the best balancing effort. In the current version, we choose the split which separates the trees with at most 40-60 bias, while minimizing the number of generated remote index nodes.

By representing the sectors as memory mapped files, persistency is achieved, since the contents of a sector can be easily dumped to, and loaded from secondary storage. Additionally, migration becomes trivial, since the persistent copy can be sent across the network, and re-mapped into another system's memory (of course, we assume endian-compatible machines).

Last but not least, by clustering the big index into continuous memory sectors, we can further reduce its size. By restricting sector sizes to 2 Gb, we can use 32bit relative pointers and further reduce the overall memory footprint of the tree. We have not been able to conduct large tests yet, however, since with 64bit pointers we could not determine any significant differences in memory consumption between the centralized and decentralized implementations, with the 32bit ones, we expect a memory reduction on the order of 40%.

5.2 Architectural Overview

Figure 8 portrays the major system components, as well as the communication flow around the cluster. There are a couple of central components listed on the figure, as well as some internal ones which are not, all of which to be explained in this section.

There are two main entry points for the application. The first one is the *RESTful Interface*, which like its centralized counterpart, is tasked with waiting on incoming requests (in the form of HTTP POSTs), reading and forwarding them. Moreover, it will also be in charge of replying to the connected client with the response to his request, once it is resolved. The *Expression Encoder* is an intermediary component that encodes the received input data (from REST requests) into the respective internal format (for either updates or queries) that is to be used throughout. This encoding is algorithmically similar to the centralized version, (see section 3.1), yet adapted for distributed communication. The second entry point in the application is through *admin clients*. These are entities that can be used to connect to the cluster and issue live commands as well as monitor state.

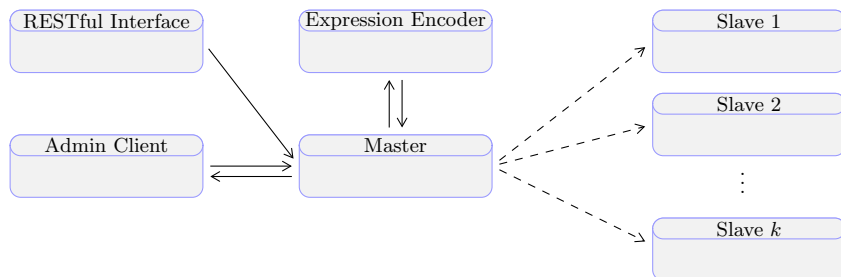


Fig. 8. Distributed Architecture - Communication Flow

Since our architecture follows a Master-Slave communication model, the main coordination in the cluster is done by the Master Node. Its responsibility is to direct external requests, from either the RESTful interface or from admin clients, to the respective slave nodes that should handle them. Furthermore, it is also tasked with internally coordinating communication and work among the various slave nodes. As such, on cluster start-up, the master comes online, prepares the internal book-keeping (including the connection to the REST daemon) and proceeds to listen for incoming connections. These can be either slaves connecting to it to join the cluster setup, or admin clients. It is important to note that, the master will only handle overview tasks – above the actual memory sector level, so it will not impose a bottleneck.

Slaves, on the other hand, are in charge of serving, updating and querying across memory sectors. Each Slave is in charge of a partition of the memory sectors, on which it performs actions upon receiving messages. These can be data messages (update or query) or admin messages (like dropping or loading specific memory sectors). While updates and queries can originate from any node, admin messages are always sent from the master node. Upon coming online, slaves need runtime information to locate the master and connect to it. After establishing connection, the master issues them a unique machine ID that will represent them throughout the session, along with some extra information to help coordinate further communication.

For this, the master maintain the following internal metadata-structures:

Slave Map represents the mapping between assigned *Slave Machine IDs* to their addresses. This is used by slaves to directly connect to each other.

Memory Sector Distribution Map represents the mapping between assigned *Memory Sector IDs* to the slave machine IDs of the servants. This is used to resolve requests that need to go to a specific memory sector directly into a message to a machine.

Expression Root Map represents the mapping between an encoded expression token ID to the memory sector id of its container⁹. This is used by the master to resolve a request on a given expression directly to a memory sector that it will be found on.

Split Tree Record represents the history of splits for all the memory sectors in the system, as well as data about the designated remote index node IDs. It maps each remote index node ID to the corresponding memory sector. This is used by the slave nodes to resolve remote index node jumps to the subsequent memory sectors. Also, this is lively updated throughout the cluster in case of memory sector splits.

Some of this metadata is also copied to the slaves: The Slave Map and Memory Sector Distribution Map are fully replicated, as slaves need to resolve requests pertaining to memory sectors that they might not be responsible for. Furthermore, the part of the Split Tree Record that related to the memory sectors a slave is responsible for is also replicated on each respective slave, for the same

⁹ The memory sector forest contains trees starting from these expression roots.

resolution reasons. The Expression Root Map is only used by the master to direct encoded requests (and it is modified on every update going towards memory sectors).

Another important observation is that, once a master-slave connection is established, this enables a two way communication protocol. Through this, the master is allowed to send messages to the slave at any given time (such as query or update requests or status notices), while the slave is responsible to periodically check-in with the master (currently, every 60 seconds) via heartbeats. For each of these heart-beats, the master will lazily¹⁰ ask the slave to update their internal mappings, should it be necessary. If a slave fails to respond within the allotted time, it will be assumed dead by the master and thus it and all mapping entries that would lead to it or its memory sectors will be invalidated.

Distributed Updates. In the current setup, the only way to load the main index of the system is via admin commands. Through this, the master takes in an MWS harvest file and proceeds to read and parse the expressions. For each one, the Expression Encoder is used to transform it into internal representation. Afterwards, if the Expression Root already exists in some memory sector, the prepared insertable item is forwarded to the respective machine that is responsible for it, by performing a lookup on the Expression Root Map and subsequently on the Memory Sector Distribution Map. If the Expression Root did not exist, than it will be assigned to a memory sector in a Round Robin fashion and then the request will be forwarded. Finally, once the request reaches the respective memory sector, a DFS of the tree starting at the Expression Root will end in two possible ways. Either a leaf index node is reached and a database insertion will be triggered, or a remote index node will be discovered, case in which, after mapping resolution, the remaining request will be forwarded to the respective machine and the algorithm recurses there.

Distributed Querying. As for queries, upon receiving a request Q , the RESTful interface assigns it an unique ID and passes it on to the Expression Encoder and then onto the master. From here, the same resolution used for insertion is employed to find the slave which serves the memory sector containing the root of the query expression. Here, the query is first searched in the slave cache. In case of a hit, the response is sent back to the RESTful interface, which uses the assigned ID to reply to the client. If the query misses the cache, the slave node initiates and coordinates the internal searching process. The expression look up starts within the memory sector. Whenever a remote index node is reached, the query is forwarded to the respective memory sector. However, when a leaf index node is reached, the respective solutions are fetched from the database, forwarded to the coordinating slave node and cached there. An individual search process is stopped either when there are no more pending remote queries, or when

¹⁰ The lazy updates allow us to not worry about invalid connections or two-way communication requests; if messages need to be routed to machines that do not have an address cached, the requests are just dropped.

a set limit of results is reached. To increase performance, remote requests are handled asynchronously, results are sent directly to the coordinator (the slave which initiated it) and queries are processed in bulks¹¹.

6 Conclusion and Future Work

We have presented a scalable extension of a search engine for mathematical formulae. In contrast to other approaches, MATHWEBSEARCH uses the full content structure of formulae and is easily extensible to other content-oriented formats. Our first evaluation shows that query times are low and essentially constant in index/harvest size, so that a search engine can scale up to web proportions. Contrary to our expectations, index size is linear in harvest size for the ARXIV corpus, which transcends the main memory limits of standard servers. Therefore, developing parallelization/distribution strategies is a priority. This paper reports the establishment of the core distribution algorithms and functionality and shows viability of the approach in principle. Exploring the distribution, management, and load balancing of a distributed cluster of search nodes is beyond the scope of this paper.

Even though based on standard practices from distributed systems, the system architecture presented here is tailored to the case of distributed substitution indexes and unification-based queries. Standard database distribution techniques do not seem to be applicable, since there indexes essentially contain metadata about locating or accessing data efficiently (and are thus orders of magnitude smaller than the data itself), whereas the substitution tree index is basically an organization of the data (formulae) itself optimized for sharing. Generic database features, such as ACID properties, which might benefit from database distribution techniques are not currently in the scope of this system.

We conclude the paper with a tabulation of open research areas for information retrieval in mathematical/technical documents.

6.1 Additional Corpora

The ARXIV corpus we are currently using for benchmarking is paradigmatic for the “informal but rigorous mathematics” that dominates mathematical communication today. Here, the Content MATHML has to be heuristically reconstructed from the presentation in the sources. This is unnecessary for corpora of formalized mathematics, e.g. the Mizar Mathematical Library [Miz] with over 50 000 formal theorems. The problems of obtaining the content MATHML are different here: Even though the representations are formal in principle, the surface languages are human-oriented, and fully explicit representations need reconstruction processes (e.g. for reconstructing elided types and arguments, resolution of operator overloading, etc.). We are currently working on Content MATHML exporters for the Mizar Library and the TPTP (Thousands of Problems for

¹¹ Although the client only requests 30 items, up to 1000 may be retrieved internally and placed into the cache.

Theorem Proving) library [SS]. Other future targets could be the input files of mathematical software systems e.g. computer algebra systems like Mathematica, numerical computation systems like MatLab, or statistics programs like the R system [Tee11].

6.2 Extending the Indexing

A current weakness of the system is the fact that it can only search for formulae that match the query terms up to α -equivalence. Many applications would for instance benefit from stronger equalities. Our search in the running example might be used to find a useful identity for $\int_0^\infty f(x) \cdot g(x) dx$, if we know that $s(x) \cdot s(x) = s^2(x)$. MATHWEBSEARCH can be extended to an *E-Retrieval* engine (i.e., search modulo an equational theory E or logical equivalence) without compromising efficiency by simply E -normalizing index and query terms (see [NK07] for a first implementation).

In the long run, we plan to extend MATHWEBSEARCH, so that it can take more document context information into account, i.e., not just keywords from the text around the formulae but e.g. the topology of theories in the OMDOC format: It would be very useful, if we could restrict searches to formulae that are consistent with current (mathematical) assumptions.

6.3 Result Ranking

Advances in ranking have made word-based search engines scalable from a user point of view. For formula search engines ranking is an open research question, there is only one paper that covers this in a more presentation-search oriented setting [You06a]. To solve the problem, we have to consider the following aspects: 1. What is a good measure for relevance in theory (pagerank only applies to pages)? 2. How can we compute this efficiently. 3. Can we organize the index, so that it finds the most relevant hits (as estimated by this measure) first? 4. Is a single measure enough? We plan to look at the following simple measures as starting points: 1. pagerank by citations over the papers 2. the size (whatever that means) of the substitutions (small might be beautiful) 3. similarly, the size of the formulae 4. popularity of the papers (by download) Finally, we would like to allow specification of content queries using more widely known formats, like L^AT_EX: strings like $\frac{1}{x^2}$ or $1/x^2$ could be processed as well. This can be reached by applying an extension (by query variables) of the L^AT_EX to XML transformation used on the ARXIV to process queries. The new L^AT_EXML daemon [GSK11] allows integrating this efficiently.

6.4 Advanced Search Services

Another important application of the unification search in MATHWEBSEARCH is applicable theorem search (like our example with Hölder's inequality in Section 2). The MATHWEBSEARCH system already supports the necessary queries

(unification), but the ARXIV corpus we are currently using does not have the necessary degree of formalization (explicitly marked up universal quantifications). We plan to utilize (possibly shallow) linguistic technologies to reliably analyze phrases like “*Let f and g be functions from \mathbb{R} to \mathbb{R} ...*” that mark the identifiers f and g as universal and to retrieve the associated sortal restrictions. Note that the linguistic capabilities of the variable spotter have to be considerable to detect the difference between “...where c is a natural number” and “...where x is the number between 1 and n , such that...” (only is c universal) or to detect that universals in a negative scope are indeed existential.

References

- [arXMLiv] arXMLiv build system, <http://arxmliv.kwarc.info> (visited on May 15, 2010)
- [Ber] Berkeley, D.B.: <http://www.oracle.com/technology/products/berkeley-db/index.html> (visited on March 03, 2010)
- [BF06] Borwein, J.M., Farmer, W.M. (eds.): MKM 2006. LNCS (LNAI), vol. 4108. Springer, Heidelberg (2006)
- [Dav+11] Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.): MKM 2011 and Calculemus 2011. LNCS (LNAI), vol. 6824. Springer, Heidelberg (2011)
- [Gra96] Graf, P.: Term Indexing. LNCS, vol. 1053. Springer, Heidelberg (1996)
- [GSK11] Ginev, D., Stamerjohanns, H., Miller, B.R., Kohlhase, M.: The LaTeXXML Daemon: Editable Math on the Collaborative Web. In: Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.) MKM 2011 and Calculemus 2011. LNCS (LNAI), vol. 6824, pp. 292–294. Springer, Heidelberg (2011)
- [Kau+07] Kauers, M., Kerber, M., Miner, R., Windsteiger, W. (eds.): MKM/CALCULEMUS 2007. LNCS (LNAI), vol. 4573. Springer, Heidelberg (2007)
- [KK07] Kohlhase, A., Kohlhase, M.: Reexamining the MKM Value Proposition: From Math Web Search to Math Web ReSearch. In: Kauers, M., Kerber, M., Miner, R., Windsteiger, W. (eds.) MKM/CALCULEMUS 2007. LNCS (LNAI), vol. 4573, pp. 313–326. Springer, Heidelberg (2007)
- [Koh+11] Kohlhase, M., et al.: The Planetary System: Web 3.0 & Active Documents for STEM. *Procedia Computer Science* 4, 598–607 (2011); Sato, M., et al. (eds.) Special issue: Proceedings of the International Conference on Computational Science (ICCS). Finalist at the Executable Papers Challenge, doi:10.1016/j.procs.2011.04.063
- [KP] Kohlhase, M., Prodescu, C.: Mathwebsearch manual. Web manual. Jacobs University
- [KŞ06] Kohlhase, M., Sucan, I.: A Search Engine for Mathematical Formulae. In: Calmet, J., Ida, T., Wang, D. (eds.) AISC 2006. LNCS (LNAI), vol. 4120, pp. 241–253. Springer, Heidelberg (2006)
- [LM06] Libbrecht, P., Melis, E.: Methods to Access and Retrieve Mathematical Content in ACTIVEMATH. In: Iglesias, A., Takayama, N. (eds.) ICMS 2006. LNCS (LNAI), vol. 4151, pp. 331–342. Springer, Heidelberg (2006)
- [Mic] GNU MicroHTTPd library (July 2011), <http://www.gnu.org/software/libmicrohttpd/> (visited on November 07, 2011)

- [Miz] Mizar mathematical library, <http://www.mizar.org/library> (visited on February 12, 2009)
- [MM06] Munavalli, R., Miner, R.: Mathfind: a math-aware search engine. In: SIGIR 2006: Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 735–735. ACM Press, New York 1148348 (2006), doi: <http://doi.acm.org/10.1145/1148170>
- [MWS] Math Web Search (January 2011), <https://trac.mathweb.org/MWS/>
- [NK07] Normann, I., Kohlhase, M.: Extended Formula Normalization for ϵ -Retrieval and Sharing of Mathematical Knowledge. In: Kauers, M., Kerber, M., Miner, R., Windsteiger, W. (eds.) MKM/CALCULEMUS 2007. LNCS (LNAI), vol. 4573, pp. 356–370. Springer, Heidelberg (2007)
- [Pos] IEEE POSIX, ISO/IEC 9945 (1988)
- [SK08] Stamerjohanns, H., Kohlhase, M.: Transforming the arXiv to XML. In: Autexier, S., Campbell, J., Rubio, J., Sorge, V., Suzuki, M., Wiedijk, F. (eds.) AISC 2008, Calculemus 2008, and MKM 2008. LNCS (LNAI), vol. 5144, pp. 574–582. Springer, Heidelberg (2008)
- [SL11] Sojka, P., Liška, M.: Indexing and Searching Mathematics in Digital Libraries. In: Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.) MKM 2011 and Calculemus 2011. LNCS (LNAI), vol. 6824, pp. 228–243. Springer, Heidelberg (2011)
- [SS] Sutcliffe, G., Sutner, C.: The TPTP problem library for automated theorem proving, <http://www.tptp.org> (visited on December 12, 2011)
- [Tee11] Teetor, P.: R Cookbook, 2nd edn. O’Reilly (2011) ISBN: 978-3486705171
- [Vei] Veillard, D.: The XML c parser and toolkit of gnome; libxml
- [You06a] Youssef, A.: Methods of Relevance Ranking and Hit-content Generation in Math Search. In: Borwein, J.M., Farmer, W.M. (eds.) MKM 2006. LNCS (LNAI), vol. 4108, pp. 393–406. Springer, Heidelberg (2006)
- [You06b] Youssef, A.M.: Roles of Math Search in Mathematics. In: Borwein, J.M., Farmer, W.M. (eds.) MKM 2006. LNCS (LNAI), vol. 4108, pp. 2–16. Springer, Heidelberg (2006)

Real Algebraic Strategies for MetiTarski Proofs

Grant Olney Passmore^{1,2}, Lawrence C. Paulson¹, and Leonardo de Moura³

¹ Computer Laboratory, University of Cambridge

² LFCS, University of Edinburgh

³ Microsoft Research, Redmond

{gp351,lp15}@cam.ac.uk, leonardo@microsoft.com

Abstract. MetiTarski [1] is an automatic theorem prover that can prove inequalities involving \sin , \cos , \exp , \ln , etc. During its proof search, it generates a series of subproblems in nonlinear polynomial real arithmetic which are reduced to true or false using a decision procedure for the theory of real closed fields (RCF). These calls are often a bottleneck: RCF is fundamentally infeasible. However, by studying these subproblems, we can design specialised variants of RCF decision procedures that run faster and improve MetiTarski's performance.

1 Introduction

MetiTarski [1] is an automatic theorem prover for special functions such as \sin , \cos , \exp and \ln , with variables ranging over the real numbers [4]. A typical problem is a universally quantified first-order formula involving inequalities between real-valued arithmetic expressions involving such functions; MetiTarski can prove many nontrivial problems in seconds, such as the following problem drawn from hybrid systems verification [6]:

$$\forall t \in (0, \infty), v \in (0, \infty) \\ ((1.565 + 0.313 v) \cos(1.16 t) + (0.01340 + 0.00268 v) \sin(1.16 t)) e^{-1.34 t} \\ - (6.55 + 1.31 v) e^{-0.318 t} + v + 10 \geq 0$$

Internally, MetiTarski is a resolution theorem prover integrated with various decision procedures for the theory of real-closed fields (RCF) [2,7,9]. MetiTarski reduces its input problem to a series of logical combinations of polynomial inequalities, which are further reduced to true or false by an RCF decision procedure. Unfortunately, the RCF decision problem is hyper-exponential in the number of variables [3]. The RCF tests typically dominate the overall processor time, and thus far the success of our methods has been limited to problems in less than six variables. In this paper, we show that by analysing the structure of

¹ MetiTarski accepts first-order formulas over equations and inequalities, with terms involving real arithmetic, including an open-ended axiomatised collection of special functions. As this theory is undecidable, MetiTarski employs heuristic methods.

the RCF subproblems generated by MetiTarski, we can design specialised variants of RCF decision procedures. In many cases, RCF ceases to be a bottleneck, and MetiTarski's improved performance extends its practical reach.

MetiTarski requires axiom files that supply upper and lower bounds for the special functions of interest. In some cases, these bounds are polynomials, typically truncated Taylor series. More often, they are rational functions (fractions of polynomials) obtained from continued fraction approximations. MetiTarski includes arithmetic simplification designed to help transform special function inequalities so as to isolate one particular special function occurrence. The general resolution procedure, augmented with this simplification, automatically identifies relevant axioms, thereby replacing the special function occurrence by a polynomial or rational function that is an upper or lower bound, as appropriate for the context in which the special function occurs. In the case of a rational function, the division operator is eliminated through use of an axiom relating division with multiplication (again chosen by the general resolution mechanism). At this point, a special function inequality has been replaced by one or more polynomial inequalities.

The integration between resolution theorem proving and an RCF decision procedure takes the form of a novel simplification rule. Resolution, like DPLL-based SAT solving, is concerned with disjunctions of literals where a literal is an atomic formula or its negation. These disjunctions of literals are called *clauses*. When MetiTarski encounters a literal consisting of a polynomial inequality, it asks an RCF decision procedure whether this literal can possibly be true, taking into account its context. Formally, MetiTarski builds a conjunction combining all known clauses that express polynomial inequalities with the negations of the other literals in the clause. If this conjunction is logically inconsistent, then the literal is equivalent to false and can be deleted from the clause [1].

MetiTarski also uses RCF decisions to discard freshly-generated clauses that express nothing new, in the sense that their polynomial content is implied by other known polynomial inequalities. This RCF-based redundancy test is not essential, but it is a powerful heuristic nevertheless because it prevents the buildup of logically superfluous but syntactically complex facts.

The search for a proof typically generates hundreds of calls to the RCF decision procedure, often with gigantic formulas. In earlier work, we used QEPCAD-B [2] with a text-based interface, and in some cases formulas given to QEPCAD-B were longer than 50,000 characters. QEPCAD-B works extremely well for univariate problems, but it deteriorates rapidly with two or three variables. In such cases, when proofs are found, hardly any processor time spent in the resolution part of the proof search, and nearly all the time is spent in QEPCAD-B. A smarter approach to RCF will allow us to tackle harder problems, and to improve the speed at which we solve problems. We describe an approach to such improvements below.

1.1 Motivating Hypotheses

The following hypotheses motivate our work:

1. By studying the structure of the sequences of RCF subproblems MetiTarski generates during its proof search, we can devise specialised RCF proof methods which outperform general “off the shelf” RCF proof methods on these sequences of RCF subproblems.
2. By making use of these specialised RCF proof methods during MetiTarski’s proof search, we can significantly improve MetiTarski’s performance.

The results in this paper strongly support these hypotheses. Moreover, extrapolating from the particular case of MetiTarski, we believe this work supports a broader hypothesis: That a methodology of studying the structure of generated subproblems and specialising decision methods to them can lead to substantial gains for many similarly arranged combinations of automatic proof methods.

1.2 Overview of Contributions

Based upon detailed analysis of the RCF subproblems generated by MetiTarski, we have made the following improvements:

1. A method for quickly recognising the satisfiability of generated \exists RCF subproblems through retaining, during any given MetiTarski run, a history of past models produced for satisfiable RCF subproblems. This improvement works because \exists RCF subproblems generated during MetiTarski proof search very often have models in common with each other. To instrument this improvement, we communicate models between MetiTarski and the external RCF decision methods it invokes. When a retained past model consists only of rational points, we test the model against new \exists RCF subproblems from within MetiTarski alone.
2. We observe that the univariate polynomials appearing in RCF subproblems generated by MetiTarski are almost always irreducible over $\mathbb{Z}[x]$. Thus, attempting to factor them, which is a step applied by default by most RCF decision methods known to us, is almost always a waste of time. For the \exists RCF decision procedure in the SMT solver Z3 [4], we observe that per RCF instance, disabling univariate factorisation has only a small speed-up, usually less than 0.02 seconds [2]. However, for typical univariate RCF subproblems, this speed-up is anywhere from 40% - 90% of the total decision method run time. As MetiTarski may generate many thousands of RCF subproblems during its search for a single proof, each of which may contain tens of different polynomials, this speed-up nontrivially aggregates, leading to serious gains.

Methodologically, these improvements were motivated by extensive computational study of the RCF subproblems generated during MetiTarski proofs [3].

² The Z3 program we use in this paper is a new unreleased version with `nlsat`, a novel approach to making \exists RCF decisions. It (and an accompanying paper [7]) may be retrieved: <http://cs.nyu.edu/~dejan/nonlinear/>

³ This collection of RCF subproblems consists of over 400,000 \exists RCF sentences, each occurring in MetiTarski’s search for a proof of one of ≈ 800 special function inequality benchmarks drawn from many mathematical, scientific and engineering sources.

The success of these methods is supported by extensive experimentation as well. As we shall see, their combination allows MetiTarski to find proofs much more quickly than it can with non-specialised “off the shelf” RCF proof methods.

2 Model Sharing

Given φ , a universally quantified boolean combination of special function inequalities, MetiTarski attempts to prove φ through a combination of resolution and RCF reasoning. For the MetiTarski problems considered in this paper, these generated RCF subproblems are always purely \exists . We will say an \exists RCF sentence F is *satisfiable* if it is true, i.e., if $\exists \mathbf{r} \in \mathbb{R}^n$ s.t. $QF(F)(\mathbf{r})$ holds, where $QF(F)$ is the quantifier-free matrix of F . We say F is *unsatisfiable* otherwise. Let F_1, \dots, F_k be the sequence of RCF subproblems generated by MetiTarski during its search for a proof of φ . Then, the following hold:

1. F_i only contributes to a MetiTarski proof when F_i is unsatisfiable over \mathbb{R}^n ,
2. Many of the F_i share common subexpressions with each other.

From the first point, we see that time spent analysing the truth of satisfiable RCF subproblems is ultimately time wasted for MetiTarski. Thus, it is desirable to have methods for quickly recognising and throwing away satisfiable F_i . Combining this desire with the second point above, we are led naturally to the following question:

Given a satisfiable RCF subproblem F_i and a subsequent satisfiable RCF subproblem F_{i+k} , is it often the case that F_i and F_{i+k} have a model in common?

As we will see, the answer to this question is a resounding *yes*. These observations lead to one of our key improvements to MetiTarski: By recording in MetiTarski models that an RCF decision procedure has found for satisfiable F_i 's, we can gain a tremendous speed-up by using these past models to quickly recognise the satisfiability of subsequently generated RCF subproblems. The overhead involved in communicating, storing and testing these models is far outweighed by the savings made through avoiding invoking an RCF decision method.

2.1 MetiTarski Proof Search in More Detail

To motivate this model-sharing improvement to MetiTarski, let us study the sequence of RCF subproblems generated during MetiTarski's search for a proof of a particularly simple special function inequality⁴. In our benchmark set, this problem is named `max-sin-2` and is the following claim over \mathbb{R} :

⁴ This problem can in fact be solved quickly by Mathematica directly, using some recent methods it contains for computing with transcendental functions [10]. We focus on this problem in our discussion of MetiTarski's proof search for didactic reasons.

$$\forall x \in (-8, 5) \quad \max(\sin(x), \sin(x + 4), \cos(x)) > 0.$$

In searching for a proof of this theorem, MetiTarski will make use of axioms it knows for \sin , \cos and \max . With default settings, and without using any of the enhancements we describe in this paper, MetiTarski finds a proof consisting of 600 steps. Each step is either a resolution step, a substitution step, an arithmetical simplification step, or an RCF decision step. This proof makes use of three different lower bounds and three different upper bounds for \cos , six different upper bounds and six different lower bounds for \sin , and two definitional axioms for \max . For example, one of the \sin lower bounds used is the following:⁵

```
cnf(sin_lower_bound_5_neg, axiom,
  (0 < X |
    ~lgen(R, Y,
      X - 1/6 * X ^ 3 + 1/120 * X ^ 5 - 1/5040 * X ^ 7 +
      1/362880 * X ^ 9 - 1/39916800 * X ^ 11 +
      1/6227020800 * X ^ 13 - 1/1307674368000 * X ^ 15 +
      1/355687428096000 * X ^ 17 - 1/121645100408832000 * X ^ 19
      + 1/51090942171709440000 * X ^ 21) | lgen(R, Y, sin(X)))).
```

Many of the intermediate clauses used in the proof contain very large polynomials with high coefficient bit-width and degree. When pretty-printed to a text file at 75 columns per line, this proof consists of 12,453 lines.

Let us now examine some properties of MetiTarski's search for this proof. The total number of RCF inferences used in the proof is 62. But how many RCF subproblems were generated and sent to an RCF decision procedure in search of this proof? This number is much higher: 2,776. Of these subproblems, 2,221 are satisfiable. Thus, over 80% of the RCF subproblems generated cannot contribute anything towards MetiTarski's proof. Deciding their satisfiability is a waste of time. This waste can be large, as the RCF subproblems are often very complicated. For instance, the set of all polynomials appearing in these 2,776 RCF subproblems has the following statistics: max total degree is 24, average total degree is 3.53, max coefficient bit-width is 103, and average coefficient bit-width is 21.03.

To get an idea of the expense involved in deciding these satisfiable RCF subproblems generated by MetiTarski, let us examine them using Mathematica's `Reduce` command. This command is one of the best and most sophisticated general-purpose tools for deciding RCF sentences, containing highly-tuned implementations of a vast array of approaches to making RCF decisions [8,9].

Using Mathematica's `Reduce` to decide all of these 2,776 RCF sentences, we see that 253.33 seconds is spent in total. Of that time, 185.28 seconds is spent deciding the satisfiable formulas. Thus, over 70% of the total RCF time for MetiTarski's proof search is spent deciding formulas which in the end can contribute nothing to MetiTarski's proof. Such results are typical. Table 1 analyses ten representative problems. For each, it displays the effort (in terms of the number of

⁵ Here `lgen(R, X, Y)` is a generalised inequality relation. It eliminates the need to have separate instances of the axiom for $<$ and \leq .

RCF problems and the time taken deciding them), followed by the subset of this effort that is wasted on satisfiable problems and finally the percentage of wasted effort, again in terms of the number of problems and the time taken. We list the contents of these problems in Table 2. Clearly, quick methods for identifying and discarding satisfiable RCF subproblems could greatly improve performance.

Table 1. RCF Subproblem Analysis for Ten Typical Benchmarks

Problem	All RCF		SAT RCF		% SAT	
	#	secs	#	secs	#	secs
CONVOI2-sincos	268	3.28	194	2.58	72%	79%
exp-problem-9	1213	6.25	731	4.11	60%	66%
log-fun-ineq-e-weak	496	31.50	323	20.60	65%	65%
max-sin-2	2776	253.33	2,221	185.28	80%	73%
sin-3425b	118	39.28	72	14.71	61%	37%
sqrt-problem-13-sqrt3	2031	22.90	1403	17.09	69%	75%
tan-1-lvar-weak	817	19.5	458	7.60	56%	39%
trig-squared3	742	32.92	549	20.66	74%	63%
trig-squared4	847	45.29	637	20.78	75%	46%
trigpoly-3514-2	1070	17.66	934	14.85	87%	84%

Now, given our previous discussions, it is natural to ask the following: How many of these satisfiable RCF subproblems share models with each other? Obtaining an exact answer to this question is certainly computationally infeasible. However, we can obtain a lower bound. We will do this in the following simple way: Whenever the RCF procedure decides a formula to be satisfiable, we will ask it to report to us a model satisfying the formula, and we will store this model within a *model history* data-structure in MetiTarski. Note that these models may in general contain irrational real algebraic points. Whenever we encounter a new RCF subproblem, we will first check, within MetiTarski, whether this RCF subproblem is satisfied by any rational model we have recorded within the model history.

Performing this experiment, we see that at least 2,172 of the 2,221 satisfiable RCF subproblems share a common model with a previously generated SAT RCF subproblem. Moreover, only 37 separate rational models were used to satisfy all of these 2,172 formulas. Note that these numbers are very much *lower* bounds, as we (i) only consider the particular models previously recorded (i.e., perhaps F_i and F_{i+k} share a model, but this common model is different than the one we have recorded for F_i), and (ii) we have only considered common rational models.

In Table 3, we show this type of model sharing analysis for the same collection of ten benchmark problems encountered previously. For each MetiTarski problem considered, we show (i) the number of SAT RCF subproblems generated, (ii) the number of those problems which could be recognised to be SAT using the simple rational model-sharing described above, (iii) the number of different rational models stored in MetiTarski’s model history, and (iv) the number of those

Table 2. Typical MetiTarski Benchmarks

CONVOI2-sincos
$\forall t \in (0, \infty), v \in (0, \infty)$
$(1.565 + 0.313v) \cos(1.16t) + (0.01340 + 0.00268v) \sin(1.16t) e^{-1.34t}$ $- (6.55 + 1.31v) e^{-0.318t} + v + 10 \geq 0$
exp-problem-9
$\forall x \in (0, \infty)$
$\frac{1 - e^{-2x}}{2x(1 - e^{-x})^2} - \frac{1}{x^2} \leq \frac{1}{12}$
log-fun-ineq-e-weak
$\forall x \in (0, 12), y \in (-\infty, \infty)$
$xy \leq \frac{1}{5} + x \ln(x) + e^{y-1}$
max-sin-2
$\forall x \in (-8, 5)$
$\max(\sin(x), \sin(x+4), \cos(x)) > 0$
sin-3425b
$\forall x \in (0, \infty), y \in (-\infty, \infty)$
$(x < y \wedge y^2 < 6) \Rightarrow \frac{\sin(y)}{\sin(x)} \leq 10^{-4} + \frac{y - \frac{1}{6}y^3 + \frac{1}{120}y^5}{x - \frac{1}{6}x^3 + \frac{1}{120}x^5}$
sqrt-problem-13-sqrt3
$\forall x \in (0, 1)$
$1.914 \frac{\sqrt{1+x} - \sqrt{1-x}}{4 + \sqrt{1+x} + \sqrt{1-x}} \leq 0.01 + \frac{x}{2 + \sqrt{1-x^2}}$
tan-1-1var-weak
$\forall x \in (0, 1.25)$
$\tan(x)^2 \leq 1.75 \cdot 10^{-7} + \tan(1) \tan(x^2)$
trig-squared3
$\forall x \in (-1, 1), y \in (-1, 1)$
$\cos(x)^2 - \cos(y)^2 \leq -\sin(x+y) \sin(x-y) + 0.25$
trig-squared4
$\forall x \in (-1, 1), y \in (-1, 1)$
$\cos(x)^2 - \cos(y)^2 \geq -\sin(x+y) \sin(x-y) - 0.25$
trigpoly-3514-2
$\forall x \in (-\pi, \pi)$
$2 \sin(x) + \sin(2x) \leq \frac{9}{\pi}$

models which were successfully shared between at least two RCF subproblems (the *successful* models in the model history). We see that with the exception of **trigpoly-3514-2**, a very large majority of the SAT RCF subproblems can be

Table 3. Model Sharing Lower Bounds for Ten Typical Benchmarks

Problem	# SAT	# SAT by MS	# \mathbb{Q} Models	# Successful
CONVOI2-sincos	194	168	9	7
exp-problem-9	731	720	11	7
log-fun-ineq-e-weak	323	305	24	18
max-sin-2	2,221	2,172	37	37
sin-3425b	72	64	8	6
sqrt-problem-13-sqrt3	1403	1350	26	21
tan-1-1var-weak	458	445	13	9
trig-squared3	549	280	15	11
trig-squared4	637	497	21	16
trigpoly-3514-2	934	4	4	2

recognised to be satisfiable through the use of past rational models. We have found the vast majority of our benchmark problems to exhibit behaviour consistent with the first nine problems in the table. We note that of those problems considered, `trigpoly-3514-2` is the only one involving π , which is approximated by MetiTarski using rational upper and lower bounds. Perhaps the presence of π in the problem has something to do with why its rational model sharing lower bounds are so much lower than the rest.

Clearly, there is much potential for improving MetiTarski through using past models of SAT RCF subproblems to quickly recognise subsequent SAT RCF subproblems. However, we have found that in some cases, the cost of finding a suitable model in our model history can be quite high. This is due to the fact that evaluating very large RCF formulas upon rational numbers of very large bit-width can become expensive (even if somewhat sophisticated approaches to polynomial sign determination are employed).

To efficiently apply this model-sharing technique in the context of MetiTarski's proof search, we have found it necessary to seek some heuristic methods for prioritising the models based upon their success rates in recognising SAT RCF subproblems. Through experimentation, we have found that prioritising models based upon *recent success* to be most useful. We store all rational models within MetiTarski, but maintain at any time a list of the ten most successful models, ordered descendingly by how recently they have been successfully applied to recognise a SAT RCF subproblem. When a new RCF subproblem is encountered, we first try the prioritised models in order. If that fails, then we try the remaining models in our model history, this time in an order based solely upon success rate.

3 Univariate Factorisations

RCF decision procedures typically devote a significant effort to factoring polynomials, effort that is wasted if a polynomial is irreducible. In our case, it has turned

out that most of the polynomials generated by MetiTarski are irreducible. This is presumably because most of the polynomials we use to bound special functions are themselves irreducible. Frequently, a bound is the ratio of two polynomials; MetiTarski will then multiply both sides by the denominator. The resulting simplifications do not necessarily have to yield another irreducible polynomial; empirically, however, this usually happens.

Of the well-known transcendental functions, polynomials involved in their bounds used by MetiTarski only have very simple factors, if they have any at all. In the case of the functions $\sin(X)$ and $\tan^{-1}(X)$, this factor is simply X , which is unsurprising because their value is zero when $X = 0$. Similarly, for the function $\ln(X)$, some polynomials have $X - 1$ as a factor. On the other hand, bounds for the function $\text{sqrt}(X)$ have many non-trivial factors. Note that the square root bounds are derived using Newton's method, while most other bounds come from Taylor series or continued fractions.

Table 4. Factorisation in RCF Subproblems for Typical Univariate Benchmarks

Problem	# Factor	# Irreducible	% Runtime
asin-8-sqrt2	7791	5975 (76.7%)	22.4%
atan-problem-2-sqrt-weakest21	65304	63522 (97.3%)	55.4%
atan-problem-2-weakest21	9882	8552 (86.5%)	2.2%
cbirt-problem-5a	88986	61068 (68.6%)	38.6%
cbirt-problem-5b-weak	138861	25107 (18.0%)	53.1%
cos-3411-a-weak	150354	138592 (92.1%)	53.9%
ellipse-check-2-weak2	5236	3740 (71.4%)	88.7%
ellipse-check-3-ln	1724	1284 (74.4%)	86.7%
ellipse-check-3-weak	12722	9464 (74.3%)	77.9%

Table 4 analyses a representative set of MetiTarski problems. For each, it displays the number of times the factorisation subprocedure is invoked in Z3, the number of times the polynomial argument is irreducible, the percentage of irreducible polynomials, and the percentage of runtime spent in the factorisation subprocedure.⁶

For univariate benchmarks, we observed that the overhead of polynomial factorisation is quite significant. Moreover, our RCF procedure in Z3 does not seem to benefit from factorisation as a preprocessing step even when polynomials can be factored. Consider the problem instances `ellipse-check-2-weak2` and `ellipse-check-3-weak` from Table 4. MetiTarski creates respectively 803 and 1569 RCF subproblems for these instances. The RCF procedure in Z3 spends respectively 88.69% and 77.95% of the runtime in the polynomial factorisation subprocedure. Although each instance can be solved in less than 20 milliseconds,

⁶ These experiments were performed on an Intel Core i7-2620M 2.70GHz with 8GB RAM running Windows 7 64-bit.

a significant amount of time can be saved by disabling the factorisation subprocedure. The experimental results in Sect. 4 demonstrate that this indeed the case.

4 Experimental Results

We have compared four separate MetiTarski runs using different RCF decision procedures: QEPCAD, Mathematica, Z3 and finally our specially modified version of Z3 incorporating the reduced factorisation strategy (cf. Sect. 3) and prioritised model-sharing (cf. Sect. 2).⁷ We have allowed up to 120 seconds per problem, using a Perl script to count how many theorems were proved in 10, 20, ..., 120 seconds processor time (the total of the time spent in proof search and RCF calls). These experiments used a subset of 409 problems taken from our full set of 853 problems. This subset omits trivial problems (defined as those that can be proved in less than one second). It also omits the existential problems, of which there are 39, because none of the new methods work for them.⁸ Figure 1 displays our results.⁹

For runtimes up to about 60 seconds, the graphs show a clear advantage for Z3 as modified using Strategy 1, but even unmodified Z3 does very well. By 120 seconds, all four runs appear to converge. This conclusion is not quite accurate, as the different decision procedures are succeeding on different problems. Mathematica does particularly well on problems with three or more variables. QEPCAD cannot prove many of these, but it does very well on univariate problems. As more processor time is allowed, Mathematica is able to prove more theorems that only it can prove, giving it an advantage.

We also compared the four decision procedures in terms of the number of problems for which they find the fastest proof. We use a threshold in this comparison, counting a proof only if it is faster by a given margin (10%, 50% or 100%, respectively) than all other proofs found; these results appear in Figure 2.

With a threshold of 10% faster, Z3 modified by Strategy 1 dramatically outperforms all other decision procedures. Its advantage decreases rapidly as this threshold is increased, while Mathematica's score largely holds steady. The situation is complicated by unique proofs: 18 theorems are proved by one system only, and of these, Mathematica proves 15. (QEPCAD-B proves one, while modified Z3 proves two.) Mathematica's superiority for higher-dimensional problems (each theorem that it uniquely proves has at least two variables, generally more) gives it an advantage as the threshold is increased, because a unique proof will

⁷ These experiments were performed on a 2×2.4 GHz Quad-Core Intel Xeon PowerMac with 10GB of 1066 MHz DDR3 RAM using QEPCAD-B 1.62, Mathematica 8.0.1 and Z3 4.0. This same machine and Mathematica installation were used for the experiments in Sect. 2.

⁸ The extension to MetiTarski allowing existentially-quantified problems must be seen as experimental. It only works on trivial problems such as $\forall y \exists x \sinh x > y$.

⁹ There is a web resource for this paper containing the MetiTarski source code, benchmarks and related data: <http://www.cl.cam.ac.uk/~gp351/cicm2012/>.

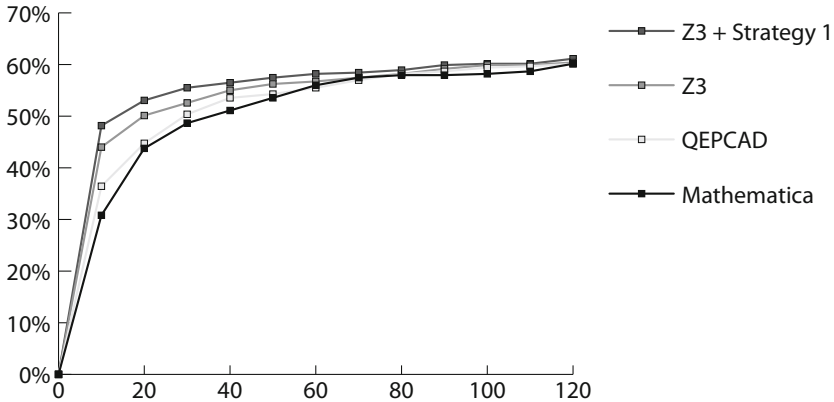


Fig. 1. Theorems Proved (by percentage of the total)

always be counted as the fastest. If the threshold is pushed high enough, only unique proofs will be counted, and here Mathematica has an inbuilt advantage. Modified Z3 remains top even with the threshold of 200% faster (which means three times faster). Mathematica finally wins at four times faster, with 17 problems against 8 for modified Z3, but these are mostly unique proofs rather than faster proofs.

Our data suggest another question: how is it that QEPCAD-B so often outperforms Mathematica, especially on univariate problems? Mathematica has much better algorithms for real algebraic numbers, and is generally more up-to-date. Overheads outside of Mathematica’s core RCF decision procedure are presumably to blame. At present, we do not know whether these overheads are concerned with parsing, preprocessing or something else altogether.

5 Future Work

We see many ways this work might be improved and extended. First, we would like to better understand how the lineage of RCF subproblems (i.e., the clauses from which the RCF subproblems were generated) influences model sharing. If two SAT RCF subproblems share a common ancestry, is it more likely that they might share a model? This seems likely. It seems plausible that lineage-based methods for prioritising which stored models we should try may yield serious efficiency improvements. It would also be very interesting to incorporate machine learning into this process. Second, we would like to make use of irrational real algebraic models in our model-sharing machinery. Currently, only rational models are used to recognise SAT RCF subproblems from within MetiTarski. One approach which interests us involves using retained real algebraic models to guide an RCF proof procedure towards certain regions of \mathbb{R}^n . This may involve combining techniques based upon interval constraint propagation and *paving* [5] to guide the manner in which Z3 explores its search space, for instance.

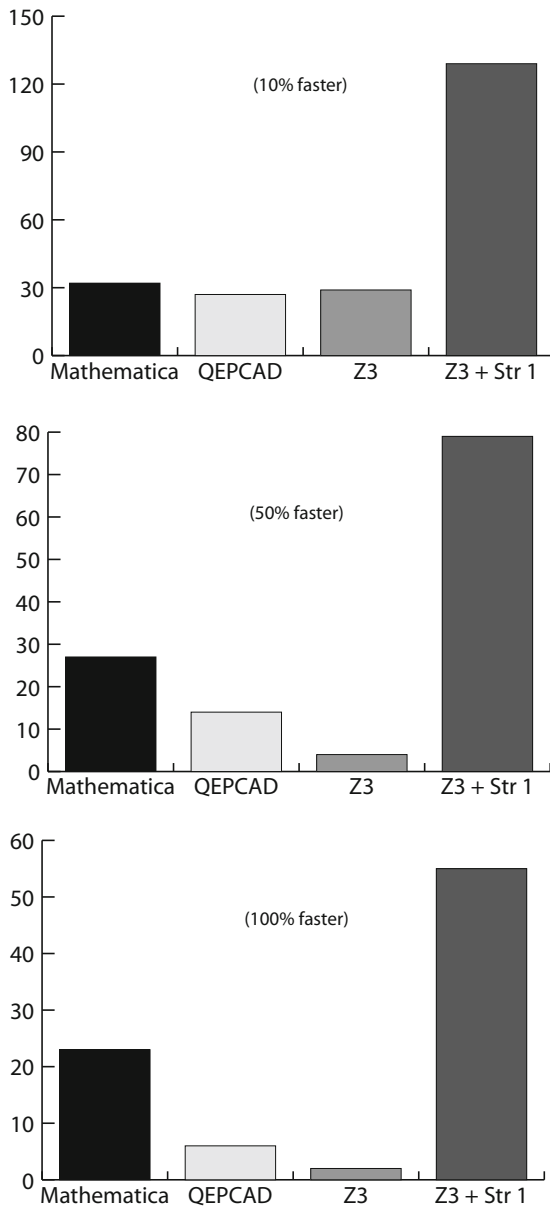


Fig. 2. Number of Fastest Proofs (by the Given Threshold) Per Run

6 Conclusion

We have shown that through detailed analysis of the RCF subproblems generated during MetiTarski's proof search, we can devise specialised variants of RCF decision procedures that greatly outperform general-purpose methods on these problems.

The approach described here is applicable to the design of any expensive proof procedure. Given a sufficiently large corpus of representative problems, the general-purpose procedure can be tuned, which should yield dramatically better results. This principle also applies when proof procedures are combined: the subsidiary proof engine should not be viewed as a black box, but should be refined by analysing the generated problems given to it. It follows that expensive proof procedures should offer easy customisation so that their users can try such refinements with the least effort.

Acknowledgements. The research was supported by the Engineering and Physical Sciences Research Council [grant numbers EP/I011005/1 and EP/I010335/1]. We thank the referees for their helpful suggestions which improved our paper.

References

1. Akbarpour, B., Paulson, L.: MetiTarski: An Automatic Theorem Prover for Real-Valued Special Functions. *Journal of Automated Reasoning* 44(3), 175–205 (2010)
2. Brown, C.W.: QEPCAD-B: A System for Computing with Semi-algebraic Sets via Cylindrical Algebraic Decomposition. *SIGSAM Bull.* 38, 23–24 (2004)
3. Davenport, J.H., Heintz, J.: Real Quantifier Elimination is Doubly Exponential. *J. Symb. Comput.* 5, 29–35 (1988)
4. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) *TACAS 2008*. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
5. Granvilliers, L., Benhamou, F.: RealPaver: An Interval Solver using Constraint Satisfaction Techniques. *ACM Trans. on Mathematical Software* 32, 138–156 (2006)
6. Gulwani, S., Tiwari, A.: Constraint-Based Approach for Analysis of Hybrid Systems. In: Gupta, A., Malik, S. (eds.) *CAV 2008*. LNCS, vol. 5123, pp. 190–203. Springer, Heidelberg (2008)
7. Jovanović, D., de Moura, L.: Solving Nonlinear Arithmetic. In: *IJCAR 2012* (2012)
8. Strzebonski, A.: Solving Systems of Strict Polynomial Inequalities. *Journal of Symbolic Computation* 29(3), 471–480 (2000)
9. Strzebonski, A.: Cylindrical Algebraic Decomposition using Validated Numerics. *Journal of Symbolic Computation* 41(9), 1021–1038 (2006)
10. Strzebonski, A.: Real Root Isolation for Tame Elementary Functions. In: *Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation, ISSAC 2009*, pp. 341–350. ACM, New York (2009)

A Combinator Language for Theorem Discovery

Phil Scott and Jacques Fleuriot

School of Informatics, University of Edinburgh, UK
phil.scott@ed.ac.uk, jdf@inf.ed.ac.uk

Abstract. We define and implement a combinator language for intermediate lemma discovery. We start by generalising an algebraic data-structure for unbounded search and then extend it to support case-analysis. With our language defined, we expect users to be able to write discoverers which collaborate intelligently in specific problem domains. For now, the language integrates rewriting, forward-deduction, and case-analysis and discovers lemmas concurrently based on an interactive proof context. We argue that the language is most suitable for adding domain-specific automation to mechanically formalised proofs written in a forward-style, and we show how the language is used via a case-study in geometry.

1 Introduction

The task of exhibiting a formal proof of a mathematical theorem is *hard*. Domain-agnostic tools are generally unusably slow at automatically finding proofs of even relatively trivial mathematical theorems. Instead, those attempting to produce formal verifications of mathematics typically use *interactive proof assistants*. Here, a human can contribute their own mathematical insight, typically by guiding and coordinating various proof searching tools in an intelligent way. But even then, the task is laborious. It has been estimated that it takes a week for an expert to formalise just one page of prose mathematics [4], which is a big problem for large scale verification efforts such as the Flyspeck Project [3].

1.1 Incidence Reasoning in the *Foundations of Geometry*

The labour involved in formalised mathematics has been of particular concern to us in regards to Hilbert's *Foundations of Geometry*. We have been formalising Hilbert's axiomatics declaratively in the LCF-style [2] theorem-prover HOL Light [5], and previous work here [7,11] has shown that when its proofs are formalised, the majority of the proof-script involves reasoning about *incidence* between points, lines and planes. This domain is clearly combinatorial, and it is challenging to find the correct paths of inference towards the necessary lemmas, or to even know what inferences are possible. The prose text is of no help, since Hilbert omits all of the complexity of incidence reasoning.

Our early proof-scripts were therefore dominated by forward inference steps that derived a large collection of collinear, non-collinear and planar sets. These, in turn, served to justify the explicit inferences in Hilbert's prose proofs.

We typically worked out the implicit incidence reasoning by hand, and then translated these into our mechanisation. Working out the correct inferential paths was laborious and error-prone, and the vast number of choices meant that in various places we found that we had chosen suboptimal paths.

Once the proofs were mechanised, there were clear patterns of reasoning, as the incidence theorems were fed through our various incidence rules. Sketching out the typical flow of incidence theorems revealed the complex network shown in Figure 1.

The aim of this paper is to express this pattern of forward-reasoning and others like it using a suitable algebraic language for intermediate lemma discovery. Following the examples of tactics and conversions [8], we implement the language in terms of ML-combinators [14]. The advantage of this implementation is that it fully integrates with the host programming language (in this case, ML [9]). This means the user is free to extend the system and is free to parameterise and inject computations into it using arbitrary pieces of the native language.

We should stress that our focus here is on *programmable* automated tools to assist *interactive* proof. As such, many of the concerns of more general theorem discovery need not apply. For instance, we will not attempt to formalise a general notion of “interestingness”. Our lemma search is implicitly guided by the proof context during interactive proof, and so interestingness can be controlled by the users who may: guide the search by refining the proof context; apply their own filters, and tailor the composition of discoverers to their chosen domain; use proof commands to reference lemmas of interest from those discovered. For more general theory discovery, see, for instance, Colton et al [1] and McCasland and Bundy [6].

In the next section, we describe the basic data-structures and combinators for a *discovery* algebra. Then, in §3, we add primitives and derive new functionality which tailor the tools towards interactive theorem proving. Finally, in §4, we evaluate our new tools in the context of the problem we have outlined here.

2 Stream Discoverers

We take it that the overarching purpose of search and discovery is to output one or more theorems. If we think of this output *as* the implementation, then we can unify both search and discovery in terms of a procedure which lazily generates successive elements of a list. Search and discovery are distinguished only according to whether we expect the lists to be finite or whether we expect them to be infinite *streams*.

For the purposes of this section, we leave unspecified what computations are used to generate the basic streams. It might be that a stream is produced by supplying it with a list of definite theorems; it might be that a stream is generated using input data; it might even be delivered by some other automated tool. We shall focus instead on the *algebra* of transformations on streams, and in how we might lift typical symbolic manipulation used in theorem proving to the level of streams.

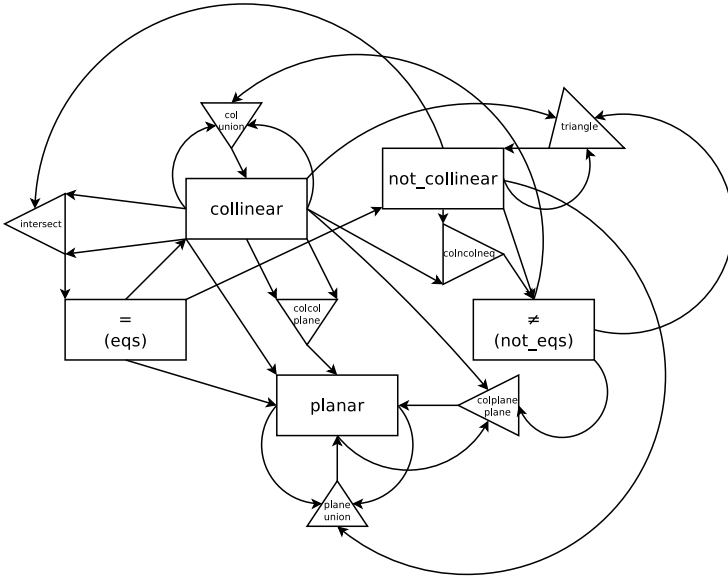


Fig. 1. A network for incidence reasoning: boxes represent collections of various kinds of theorem, while triangles represent inference rules

One reason why lists and streams are a good choice is that they are *the* ubiquitous data-structure of ML and its dialects, and have rich interfaces to manipulate them. A second reason why lists are an obvious choice is that they have long been known to satisfy a simple set of algebraic identities and thus to constitute a monad [15]. We can interpret this monad as decorating computations with non-deterministic choice and backtracking search.

Monads themselves have become a popular and well-understood abstraction in functional programming, and have gained recent interest in other languages. Formally, a monad is a type-constructor M together with three operations

$$\begin{aligned}
 \text{return} &: \alpha \rightarrow M \alpha \\
 \text{fmap} &: (\alpha \rightarrow \beta) \rightarrow M \alpha \rightarrow M \beta \\
 \text{join} &: M (M \alpha) \rightarrow M \alpha
 \end{aligned}$$

satisfying the algebraic laws given in Figure 2.

However, the list monad uses list concatenation $\text{concat} : [[\alpha]] \rightarrow [\alpha]$ as the join, which makes it unsuitable for non-terminating *discovery*. If the list xs represents unbounded discovery, then we have $xs + ys = xs$ ¹ for any ys , and thus, all items discovered by ys are lost. This is an undesirable property. We want to be able to combine unbounded searches over infinite domains without losing any data.

¹ Here, + is just list append.

$$\begin{aligned}
 \text{fmap } (\lambda x. x) m &= m \\
 \text{fmap } f \circ \text{fmap } g &= \text{fmap } (f \circ g) \\
 \text{fmap } f \circ \text{return} &= \text{return} \circ f \\
 \text{fmap } f \circ \text{join} &= \text{join} \circ \text{fmap } (\text{fmap } f) \\
 (\text{join} \circ \text{return}) m &= m \\
 (\text{join} \circ \text{fmap } \text{return}) m &= m \\
 \text{join} \circ \text{join} &= \text{join} \circ \text{fmap } \text{join}
 \end{aligned}
 \tag{1}$$

Fig. 2. The Monad Laws

2.1 The Stream Monad

There is an alternative definition of the monad for streams (given in Spivey [13]) which handles unbounded search. Here, the join function takes a possibly infinite stream of possibly infinite streams, and produces an exhaustive enumeration of *all* elements. We show how to achieve this in Figure 3 using a function `shift`, which moves each stream one to the “right” of its predecessor. We can then exhaustively enumerate every element, by enumerating each column, one-by-one, from left-to-right.

$$\begin{aligned}
 & \text{shift} \begin{bmatrix} [D_{0,0}, D_{0,1}, D_{0,2}, \dots, D_{0,n}, \dots], \\ [D_{1,0}, D_{1,1}, D_{1,2}, \dots, D_{1,n}, \dots], \\ [D_{2,0}, D_{2,1}, D_{2,2}, \dots, D_{2,n}, \dots], \\ [D_{3,0}, D_{3,1}, D_{3,2}, \dots, D_{3,n}, \dots], \\ [D_{4,0}, D_{4,1}, D_{4,2}, \dots, D_{4,n}, \dots], \\ [D_{5,0}, D_{5,1}, D_{5,2}, \dots, D_{5,n}, \dots], \end{bmatrix} \\
 &= \begin{bmatrix} [D_{0,0}, D_{0,1}, D_{0,2}, D_{0,3}, D_{0,4}, D_{0,5}, D_{0,6}, D_{0,7}, D_{0,8}, D_{0,9}, D_{0,10}, \dots], \\ [D_{1,0}, D_{1,1}, D_{1,2}, D_{1,3}, D_{1,4}, D_{1,5}, D_{1,6}, D_{1,7}, D_{1,8}, D_{1,9}, \dots], \\ [D_{2,0}, D_{2,1}, D_{2,2}, D_{2,3}, D_{2,4}, D_{2,5}, D_{2,6}, D_{2,7}, D_{2,8}, \dots], \\ [D_{3,0}, D_{3,1}, D_{3,2}, D_{3,3}, D_{3,4}, D_{3,5}, D_{3,6}, D_{3,7}, \dots], \\ [D_{4,0}, D_{4,1}, D_{4,2}, D_{4,3}, D_{4,4}, D_{4,5}, D_{4,6}, \dots], \\ [D_{5,0}, D_{5,1}, D_{5,2}, D_{5,3}, D_{5,4}, D_{5,5}, \dots], \\ [D_{6,0}, D_{6,1}, D_{6,2}, D_{6,3}, D_{6,4}, \dots] \\ \vdots \end{bmatrix}
 \end{aligned}$$

Fig. 3. Shifting

If we understand these streams as the outputs of discoverers, then the outer stream can be understood as the output of a discoverer which *discovers discoverers*. The join function can then be interpreted as *forking* each discoverer at

the point of its creation and combining the results into a single discoverer. The highlighted column in Figure 3 is this combined result: a set of values generated *simultaneously* and thus having no specified order (this is required to satisfy Law 1 in Figure 2).

However, this complicates our stream type, since we now need additional inner structure to store the combined values. We will refer to instances of this inner structure as *generations*, each of which is a finite collection of simultaneous values discovered at the same level in a breadth-first search. We then need to define the join function, taking care of this additional structure.

Suppose that generations have type $G \alpha$ where α is the element type. The manner in which we will define our shift and join functions on discoverers of generations assumes certain algebraic laws on them: firstly, they must constitute a monad; secondly, they must support a sum operation $(+) : G \alpha \rightarrow G \alpha \rightarrow G \alpha$ with identity $0 : G \alpha$. The join function for discoverers must then have type $[G [G \alpha]] \rightarrow [G \alpha]$, sending a discoverer of generations of discoverers into a discoverer of generations of their data.

To see how to define this join function, we denote the k^{th} element of its argument by $gs_k = \{d_{k,0}, d_{k,1}, \dots, d_{k,n}\} : G [G \alpha]$. Each $d_{k,i}$ is, in turn, a discoverer stream $[g_{i,0}^k, g_{i,1}^k, g_{i,2}^k, \dots] : [G \alpha]$. We invert the structure of gs_k using a function **transpose** : $M[\alpha] \rightarrow [M \alpha]$, which we can define for arbitrary monads M . This generality allows us to abstract away from Spivey’s bags and consider more exotic inner data-structures. We choose the name “**transpose**” since its definition generalises matrix transposition on square arrays (type $[[\alpha]] \rightarrow [[\alpha]]$):

$$\text{transpose } xs = \text{fmap head } xs :: \text{transpose (fmap tail } xs)$$

The transpose produces a stream of generations of generations (type $[G (G \alpha)]$). If we join each of the elements, we will have a stream $[D_{k,0}, D_{k,1}, D_{k,2}, \dots] : [G \alpha]$ (see Figure 4), and thus, the shift function of Figure 3 will make sense. Each row is shifted relative to its predecessor by prepending the 0 generation, and the columns are combined by taking their sum.

The type of discoverers now constitutes a monad (see Spivey [13] for details). The fact that we have a monad affords us a tight integration with the host language in the following sense: we can lift arbitrary functions in the host language to functions on discoverers, and combine one discoverer $d : [G \alpha]$ with another discoverer $d' : \alpha \rightarrow [G \alpha]$ which depends, via arbitrary computations, on each individual element of d .

There is further algebraic structure in the form of a monoid: streams can be summed by summing corresponding generations, an operation whose identity is the infinite stream of empty generations.

2.2 Case-Analysis

Our algebra allows us to partition our domain into discoverers according to our own insight (for instance, in geometry, we saw we should divide the domain into various sorts of incidence relations). We can then compose the discoverers in a way that reflects the typical reasoning patterns found in the domain.

$$\begin{aligned}
 & \text{map join (transpose } \{d_{k,0}, d_{k,1}, \dots, d_{k,n}\}) \\
 = & \text{map join } \left(\text{transpose } \left\{ \begin{array}{l} [g_{0,0}^k, g_{0,1}^k, g_{0,2}^k, \dots] \\ [g_{1,0}^k, g_{1,1}^k, g_{1,2}^k, \dots] \\ \vdots \\ [g_{n,0}^k, g_{n,1}^k, g_{n,2}^k, \dots] \end{array} \right\} \right) \\
 = & \left[\begin{array}{l} \text{join } \{g_{0,0}^k, g_{1,0}^k, \dots, g_{n,0}^k\}, \\ \text{join } \{g_{0,1}^k, g_{1,1}^k, \dots, g_{n,1}^k\}, \\ \text{join } \{g_{0,2}^k, g_{1,2}^k, \dots, g_{n,2}^k\}, \\ \vdots \end{array} \right] \\
 = & [D_{k,0}, D_{k,1}, D_{k,2}, \dots]
 \end{aligned}$$

Fig. 4. Transpose

However, when it comes to theorem-proving, the sets of theorems are further partitioned by branches on disjunctive theorems. In proof-search, when we encounter a disjunction, we will want to branch the search and associate discovered theorems in each branch with its own disjunctive hypothesis.

Ideally, we want to leave such case-splitting as a book-keeping issue in our algebra, and so integrate it into the composition algorithm. Streams must then record a context for all of their theorems, and this context must be respected as discoverers are combined.

Luckily, we left ourselves room to implement the generations output by our discoverers. To solve the problem of case-analysis, we have chosen to implement the generations as *trees*. We have briefly described a version of this data-structure elsewhere [12]. However, the data-structure has since been simplified and we have now provided a definition of its `join`.

2.3 Trees

Each tree represents a discovered generation. Each node is a (possibly empty) conjunction of theorems discovered in that generation. Branches correspond to case-splits, with each branch tagged for the disjunct on which case-splitting was performed. The branch labels along any root-path therefore provide a context of disjunctive hypotheses for that subtree.

Thus, the tree in Figure 5 can be thought as representing the formula:

$$\begin{aligned}
 & \phi_1 \wedge \phi_2 \wedge \dots \wedge \phi_n \wedge (P \rightarrow \psi_1 \wedge \psi_2 \wedge \dots \wedge \psi_n) \wedge (Q \rightarrow \chi_1 \wedge \chi_2 \wedge \dots \wedge \chi_n \\
 & \quad \wedge (R \rightarrow \alpha_1 \wedge \alpha_2 \wedge \dots \wedge \alpha_n) \wedge (S \rightarrow \beta_1 \wedge \beta_2 \wedge \dots \wedge \beta_n))
 \end{aligned}$$

The principal operation on trees is a sum function which is analogous to the append function for lists, combining all values from two trees. We combine

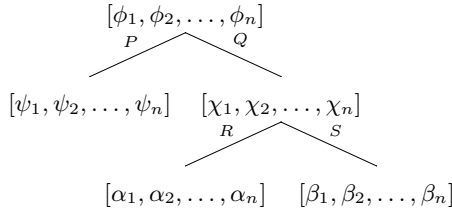


Fig. 5. Tagged Proof-trees

case-analyses by nesting them, replacing the leaf nodes of one tree with copies of the other tree. For definiteness, we always nest the right tree in the left.

To keep the trees from growing indefinitely, we consider the following simplifications: firstly, we prune subtrees which contain no data; secondly, if a case is introduced whose context is larger than a parallel case further up the tree, it can be dropped, since any theorem which can be eliminated in the stronger branch will have to be discovered again under weaker assumptions further up the tree; finally, if a branch label is introduced which appears further up the root path, then all sibling branches are dropped, while the subtree is summed with its parent — a move which corresponds to weakening. We illustrate these rules in Figure 6.

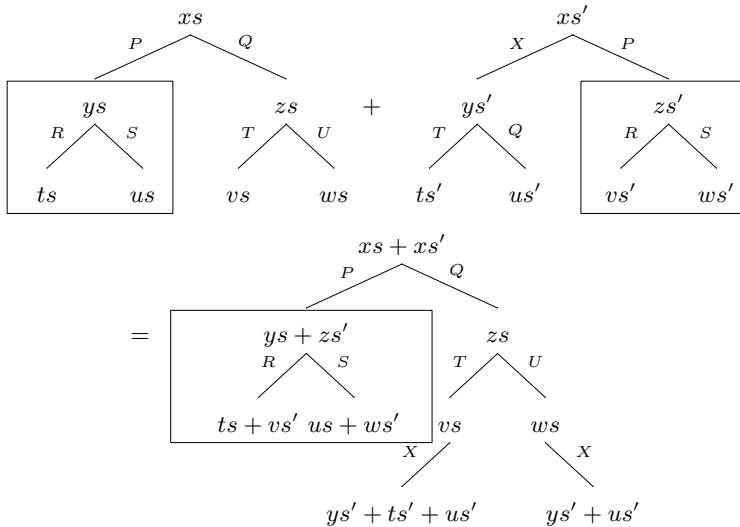


Fig. 6. Proof tree combination and simplification. The highlighted subtrees are combined applying simplification rules which yield a subtree with the same topology.

Finally, we allow trees to be empty, containing a single empty root node. This is the identity of our sum function. From these, we can define a *join* analogous to list concatenation. Suppose we are given a tree t whose nodes are themselves trees (so the type is $\mathbf{Tree} (\mathbf{Tree} \alpha)$). Denote the inner trees by $t_0, t_1, t_2, \dots, t_n : \mathbf{Tree} \alpha$. We now replace every node of t with an empty conjunction, giving a new tree t' with the *new* type $\mathbf{Tree} \alpha$. We can now form the sum

$$t' + t_0 + t_1 + t_2 + \dots + t_n$$

The resulting tree will then contain discovered theorems which respect disjunctive hypotheses from their place in t and from their respective inner-trees.

3 Additional Primitives and Derived Discoverers

As we described in §2.1, discovery constitutes a monoid. The sum function can be understood as a simple way to compose two discoverers by effectively running them in parallel, respecting their case-splits. Composing with the identity discoverer, which generates nothing but empty generations, yields the same discoverer as before.

3.1 Case-Splitting

Case-splits are introduced by `disjuncts`, which is a discoverer parameterised on arbitrary theorems. Here, `disjuncts` ($\vdash P_0 \vee P_1 \vee \dots \vee P_n$) outputs a single tree with n branches from the root node. The i^{th} branch is labelled with the term P_i and contains the single theorem $P_i \vdash P_i$. This process can be undone by flattening trees using `flatten`, which discharges all tree labels and adds them as explicit antecedents on the theorems of the respective subtrees.

3.2 Filtering

In many cases, we will not be interested in all the outputs generated by a discoverer. Fortunately, filtering is a library function for monads with a zero element, and can be defined as:

$$\begin{aligned} xs \gg= f &= \mathbf{join} (\mathbf{fmap} f xs) \\ \mathbf{filter} p xs &= xs \gg= (\lambda x. \mathbf{if} p x \mathbf{then} \mathbf{return} x \mathbf{else} 0) \end{aligned}$$

More challenging is a function to perform something akin to *subsumption*. The idea here is that when a theorem is discovered which “trivially” entails a later theorem, the stronger theorem should take the place of the weaker. This is intended only to improve the performance of the system by discarding redundant search-paths.

We can generalise the idea to arbitrary data-types, and parameterise the filtering by any partial-ordering on the data, subject to suitable constraints. One intuitive constraint is that a stronger item of data should only replace a weaker

item so long as we don't "lose" anything from later discovery. Formally, we require that any function f used as the first argument to `fmap` is monotonic with respect to the partial-order. That is, if $x \leq y$ then $f x \leq f y$.

We then implement a "subsumption" function in the form of the transformation `maxima`. This transforms a discoverer into one which does two things: firstly, it transforms every individual generation into one containing only maxima of the partial-order. Secondly, it discards data in generations that is strictly weaker than some item of data from an earlier generation. Note that the partial-ordering is automatically refined to cope with case-splits, so that data higher up a case-splitting tree is always considered stronger than data below it (since it carries fewer case-splitting assumptions).

3.3 Deduction

Direct deduction, or Modus Ponens, is the basis of forward-chaining and we provide two main ways to reproduce it for discoverers. In our theorem-prover, the Modus Ponens inference rule can throw an exception, so we first redefine `fmap` to filter thrown exceptions out of the discovery. Then we can define functions `fmap2'` and `fmap3'` which lift two and three-argument functions up to the level of discoverers.

```
fmap' f xs = xs >>= (\x. try return (f x) with _ -> 0) xs
fmap2' f xs ys = fmap f xs >>= (\f. fmap' f ys)
fmap3' f xs ys zs = fmap f xs >>= (\f. fmap2' f ys zs)
```

With these, we can define the following forward-deduction functions:

```
chain1 imp xs = fmap2' MATCH_MP (return imp) xs
chain2 imp xs ys = fmap2' MATCH_MP (chain1 imp xs) ys
chain3 imp xs ys zs = fmap2' MATCH_MP (chain2 imp xs ys) zs
chain imps xs = imps >>= (\imp. if is_imp imp
    then chain (fmap (MATCH_MP imp) thms) thms
    else return imp)
```

The function `is_imp` returns true if its argument is an implication, while `MATCH_MP imp` is a function which attempts to match the antecedent of `imp` with its argument. Thus, `chain1` applies a rule of the form $P \rightarrow Q$ across a discoverer of antecedents. The function `chain2` applies a rule of the form $P \rightarrow Q \rightarrow R$ across two discoverers of antecedents. The function `chain3` applies a rule of the form $P \rightarrow Q \rightarrow R \rightarrow S$ across three discoverers of antecedents. The final, more general function, *recursively* applies rules with arbitrary numbers of curried antecedents from the discoverer `imps` across all possible combinations of theorems from the discoverer `xs`.

3.4 Integration

Finally, we consider how our discoverers integrate with term-rewriting, declarative proof and tactics. Integrating term-rewriting is trivial: we simply lift rewriting functions with `fmap` and its derivatives.

To use our discoverers in declarative proofs, we introduce a keyword `obviously`. This keyword can be used to augment any step in a declarative proof and takes an arbitrary function mapping discoverers to discoverers. When the keyword is encountered in the evaluation of a proof script, all intermediate facts inferred up to that point in the proof are fed into the function, and search is evaluated to a maximum depth. Afterwards, discovered facts are added as justification for the augmented step.

Tactics can also readily make use of discovery. We provide a tactic `chain_tac` which, again, takes a function mapping discoverers to discoverers, and also takes a tactic which depends on a list of theorems. The tactic `chain_tac` takes a goal, feeds its hypotheses through the discovery function and evaluates search to a maximum depth. The discovered theorems are then fed to the list dependent tactic, which attempts to simplify the goal. For example, the tactic

```
chain_tac by_incidence REWRITE_TAC
```

feeds a goal's hypotheses into an incidence discoverer, and then rewrites the goal using the discovered theorems.

We finally supply a primitive discoverer `monitor` which discovers theorems concurrently during proof development. We have made this discoverer the basis for a more *collaborative* discovery framework, which we describe elsewhere [12].

4 The Problem Revisited

We now return to our original geometry problem. In Figure 7, we capture the complex network from Figure 1. Each of the five kinds of theorem depicted corresponds to the definition of a new discoverer, and the mutual dependencies of the network are captured by mutual recursion 2.

Search can now be further refined. For instance, the network in Figure 1 has some redundancy: point-inequalities delivered from non-collinear sets cannot be used to infer *new* non-collinear sets. This redundancy can be eliminated by splitting `neqs` into two discoverers, `neqs` and `neqs'`. The latter is used only to derive non-collinear sets, while the sum of both is used for all other inference.

4.1 Results

A relatively simple description of a discoverer can now systematically recover the implicit incidence-reasoning in Hilbert's *Foundations of Geometry*. We show its results through part of an example proof. Here, we are trying to prove a transitivity property of Hilbert's three-place *between* relation on points: if B lies between A and C , and C between B and D , then C lies between A and D .

² The inference rule `CONJUNCTS` sends a conjunctive theorem to list of its conjuncts.


```

sum = foldr (+) 0 ◦ map return

by_incidence thms =
  let rec collinear = maxima (filter is_collinear thms
    + fmap3' col_union (delay collinear) (delay collinear) neqs)
  and non_collinear = maxima (filter is_non_collinear thms
    + fmap3' triangle collinear (delay non_collinear) neqs)
  and eqs = filter is_eq thms
    + maxima(sum (fmap3' intersect
      collinears collinear non_collinear))
  and neqs = maxima(filter is_neq thms
    + sum (fmap2' colncolneq collinear (delay non_collinear))
    + sum (fmap' CONJUNCTS (rule1 ncol_neq non_collinear)))
  and planes = maxima (filter is_plane thms
    + fmap3' plane_union (delay planes) (delay planes)
      non_collinear
    + fmap3' colplaneplane collinear (delay planes) neqs
    + fmap2' colcolplane collinear collinear
    + fmap' colplane collinear
    + fmap' ncolplane non_collinear)
  in collinear+non_collinear + eqs + neqs + planes

```

Fig. 7. Incidence Discovery

prove between A B C \wedge between B C D \implies between A C D
 assume between A B C \wedge between B C D at 0 consider E such that such
 that $\neg(\exists a. A \text{ on } a \wedge B \text{ on } a \wedge E \text{ on } a)$ from 0 by II,1 and triangle

The **assume** step adds the goal's antecedent to the current hypotheses, while the **consider** step introduces a non-collinear point E using one of Hilbert's axioms and a lemma **triangle** (see Appendix [A](#)). These hypotheses form the context for our discoverer. They are automatically picked up by **monitor** and then fed through our incidence network to produce the following theorems within 0.31 seconds³

$A \neq E, B \neq E, \text{ between } A B C, \text{ between } B C D, A \neq B, A \neq C, B \neq C, B \neq D,$
 $C \neq D, \neg(\exists a. A \text{ on } a \wedge B \text{ on } a \wedge E \text{ on } a), \neg(\exists a. A \text{ on } a \wedge C \text{ on } a \wedge E \text{ on } a),$
 $\neg(\exists a. B \text{ on } a \wedge C \text{ on } a \wedge E \text{ on } a), C \neq E,$
 $(\exists a. A \text{ on } a \wedge B \text{ on } a \wedge C \text{ on } a \wedge D \text{ on } a),$
 $\neg(\exists a. B \text{ on } a \wedge D \text{ on } a \wedge E \text{ on } a), \neg(\exists a. C \text{ on } a \wedge D \text{ on } a \wedge E \text{ on } a),$
 $(\exists \alpha. A \text{ on } \alpha \wedge B \text{ on } \alpha \wedge C \text{ on } \alpha \wedge D \text{ on } \alpha \wedge E \text{ on } \alpha), D \neq E$

The obviously keyword picks up these theorems, and from $C \neq E$ we are able to find a point F :

obviously by_incidence consider F such that between C E F by II,2 at 1

³ We have tested this on an Intel Core 2 2.53GHz machine.

The next set of discovered theorems are found within 1.21 seconds:

between C E F, $(\exists a. C \text{ on } a \wedge E \text{ on } a \wedge F \text{ on } a)$, $C \neq F$, $E \neq F$,
 $(\exists \alpha. A \text{ on } \alpha \wedge B \text{ on } \alpha \wedge C \text{ on } \alpha \wedge D \text{ on } \alpha \wedge E \text{ on } a \wedge F \text{ on } a)$,
 $A \neq F$, $B \neq F$, $D \neq F$
 $\neg(\exists a. A \text{ on } a \wedge C \text{ on } a \wedge F \text{ on } a)$, $\neg(\exists a. A \text{ on } a \wedge E \text{ on } a \wedge F \text{ on } a)$,
 $\neg(\exists a. B \text{ on } a \wedge C \text{ on } a \wedge F \text{ on } a)$, $\neg(\exists a. B \text{ on } a \wedge E \text{ on } a \wedge F \text{ on } a)$,
 $\neg(\exists a. C \text{ on } a \wedge D \text{ on } a \wedge F \text{ on } a)$, $\neg(\exists a. D \text{ on } a \wedge E \text{ on } a \wedge F \text{ on } a)$,
 $\neg(\exists a. A \text{ on } a \wedge B \text{ on } a \wedge F \text{ on } a)$, $\neg(\exists a. B \text{ on } a \wedge D \text{ on } a \wedge F \text{ on } a)$,

The rest of the proof consists of repeatedly applying a complex axiom due to Pasch. The axiom says that if a line enters one side of a triangle then it leaves by one of the other two sides. By cleverly applying this axiom, it is possible to prove our original theorem (this is not a trivial matter, and the proof had eluded Hilbert in the first edition of *Foundations of Geometry* where the theorem was an axiom; the proof was later supplied by Moore [10]).

The challenge, however, lies in verifying when all the preconditions on Pasch's Axiom have been met, something we handle by adding a `discover_by_pasch` to our existing incidence discovery (we omit the definition for space). It reveals the following additional theorems, found within 2.82 seconds.

$\exists G. (\exists a. B \text{ on } a \wedge E \text{ on } a \wedge G \text{ on } a) \wedge (\text{between } A G C \vee \text{between } A G F)$
 $\exists G. (\exists a. A \text{ on } a \wedge E \text{ on } a \wedge G \text{ on } a) \wedge (\text{between } B G C \vee \text{between } B G F)$
 $\exists G. (\exists a. B \text{ on } a \wedge E \text{ on } a \wedge G \text{ on } a) \wedge (\text{between } A G F \vee \text{between } C G F)$
 $\exists G. (\exists a. B \text{ on } a \wedge E \text{ on } a \wedge G \text{ on } a) \wedge (\text{between } C G D \vee \text{between } D G F)$
 $\exists G. (\exists a. B \text{ on } a \wedge F \text{ on } a \wedge G \text{ on } a) \wedge (\text{between } A G E \vee \text{between } C G E)$
 $\exists G. (\exists a. D \text{ on } a \wedge E \text{ on } a \wedge G \text{ on } a) \wedge (\text{between } B G C \vee \text{between } B G F)$

Further exploration of the proof involves applying one of these theorems. We can, for instance, try the penultimate instance with the step

obviously `by_pasch` consider G such that $(\exists a. B \text{ on } a \wedge F \text{ on } a \wedge G \text{ on } a) \wedge (\text{between } A G E \vee \text{between } C G E)$

The monitor now picks up the disjunction and creates a tree to represent a case-split. As this feeds into the discoverer, our combinators will automatically partition the search on the two assumptions. Our discoverer then produces three sets of theorems in 9.58 seconds⁴

The first set of theorems are proven independently of the case-split:

$(\exists \alpha. A \text{ on } \alpha \wedge B \text{ on } \alpha \wedge C \text{ on } \alpha \wedge D \text{ on } \alpha \wedge E \text{ on } a \wedge F \text{ on } a \wedge G \text{ on } a)$
 $C \neq G \wedge E \neq G \wedge A \neq G \wedge D \neq G$

⁴ Note that the outputs from the three sets are interleaved, and are not generated simultaneously. While the full set of theorems requires 9.58 seconds, most of the theorems shown here are actually generated in under 1 second. The theorem required to advance the proof, $\text{between } C G E \rightarrow F = G$ is generated in 6.19 seconds.

The next set of theorems are discovered in a branch on the assumption of **between A G E**:

between A G E, $(\exists a. A \text{ on } a \wedge G \text{ on } a \wedge E \text{ on } a)$, $B \neq G$, $F \neq G$,
 $\neg(\exists a. A \text{ on } a \wedge B \text{ on } a \wedge G \text{ on } a)$, $\neg(\exists a. B \text{ on } a \wedge E \text{ on } a \wedge G \text{ on } a)$
 $\neg(\exists a. A \text{ on } a \wedge C \text{ on } a \wedge G \text{ on } a)$, $\neg(\exists a. C \text{ on } a \wedge E \text{ on } a \wedge G \text{ on } a)$
 $\neg(\exists a. E \text{ on } a \wedge F \text{ on } a \wedge G \text{ on } a)$, $\neg(\exists a. B \text{ on } a \wedge D \text{ on } a \wedge G \text{ on } a)$
 $\neg(\exists a. C \text{ on } a \wedge D \text{ on } a \wedge G \text{ on } a)$, $\neg(\exists a. C \text{ on } a \wedge F \text{ on } a \wedge G \text{ on } a)$
 $\neg(\exists a. D \text{ on } a \wedge F \text{ on } a \wedge G \text{ on } a)$

The final set of theorems are discovered in a branch on the assumption of **between C G E**:

between C G E, $(\exists a. C \text{ on } a \wedge E \text{ on } a \wedge F \text{ on } a \wedge G \text{ on } a)$, $B \neq G$,
 $\neg(\exists a. A \text{ on } a \wedge C \text{ on } a \wedge G \text{ on } a)$, $\neg(\exists a. A \text{ on } a \wedge E \text{ on } a \wedge G \text{ on } a)$
 $\neg(\exists a. B \text{ on } a \wedge C \text{ on } a \wedge G \text{ on } a)$, $\neg(\exists a. B \text{ on } a \wedge E \text{ on } a \wedge G \text{ on } a)$
 $\neg(\exists a. C \text{ on } a \wedge D \text{ on } a \wedge G \text{ on } a)$, $\neg(\exists a. D \text{ on } a \wedge E \text{ on } a \wedge G \text{ on } a)$
 $\neg(\exists a. A \text{ on } a \wedge B \text{ on } a \wedge G \text{ on } a)$, $\neg(\exists a. B \text{ on } a \wedge D \text{ on } a \wedge G \text{ on } a)$
 $F = G$, **between C F E**

The obviously step collapses the stream of trees, pushing the branch labels into the theorems as antecedents, and then uses the resulting lemmas to justify the step. Thus, the fact $F = G$ becomes **between C G E** $\rightarrow F = G$. This fact is sufficient to derive a contradiction with **between C E F** and thus eliminate the case-split:

obviously by `incidence` have **between A G E** from 1 by II,3

The rest of the proof proceeds similarly. While the prose proof has 9 steps and our earlier formalisation without discovery runs to over 80 steps, the new formalisation has just 17 steps. We found this roughly 80% reduction in proof length across all 18 theorems from our earlier formalisation, with the new formalisations comparing much more favourably with the prose.

5 Conclusion and Further Work

We have implemented a lemma discovery language which copes with complex interdependencies between different kinds of theorems. We demonstrated the composability with a prototype discoverer which reasons about incidence in geometry, and we showed its results when exploring a non-trivial proof.

As mentioned in earlier work [12], the language does not yet provide functions for more powerful first-order and higher-order reasoning. For instance, we would like to be able to speculate inductive hypotheses and infer universals by induction. Since the basic discovery data-type is polymorphic and not specific to theorem-proving, we hope that lemma speculation will just be a matter of defining appropriate search strategies. We would also like to handle existential reasoning automatically, and we are still working on a clean way to accomplish this.

With more case-studies, we hope to find new abstractions and derived transformations to handle such reasoning, and find ways to make it easier to write discoverers. We also would like to investigate performance issues. For now, there are inefficiencies when it comes to subsumption, and we have not yet found an effective way to integrate normalisation with respect to derived equalities. We suspect we will need to enrich the underlying data-structures to cope with this.

Acknowledgements. We would like to thank our anonymous reviewers for their excellent comments and feedback. This research was partly supported by EPSRC grant EP/J001058/1.

A Axioms and Theorems

An elementary axiom and two theorems used for incidence and order reasoning in the *Foundations of Geometry*:

Axiom II,1: $\text{between } A \ B \ C \rightarrow (\exists a. A \text{ on } a \wedge B \text{ on } a \wedge C \text{ on } a)$

$\wedge A \neq B \wedge A \neq C \wedge B \neq C \wedge \text{between } C \ B \ A$

Axiom II,2: $A \neq B \rightarrow \exists C. \text{between } A \ B \ C$

Axiom II,3: $\text{between } A \ B \ C \rightarrow \neg \text{between } A \ C \ B$

triangle: $A \neq B \rightarrow \exists C. \neg(\exists a. A \text{ on } a \wedge B \text{ on } a \wedge C \text{ on } a)$

ncolneq: $\forall A \ B \ C. \neg(\exists a. A \text{ on } a \wedge B \text{ on } a \wedge C \text{ on } a) \rightarrow A \neq B \wedge A \neq C \wedge B \neq C$

References

1. Colton, S., Bundy, A., Walsh, T.: On the notion of interestingness in automated mathematical discovery. *Int. J. Hum.-Comput. Stud.* 53(3), 351–375 (2000)
2. Gordon, M.: From LCF to HOL: a short history. In: *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pp. 169–185. MIT Press, Cambridge (2000)
3. Hales, T.: Introduction to the Flyspeck Project, <http://drops.dagstuhl.de/opus/volltexte/2006/432/pdf/05021.HalesThomas.Paper.432.pdf>
4. Hales, T.: Formal proof. *Notices of the American Mathematical Society* 55, 1370–1381 (2008)
5. Harrison, J.: HOL Light: a Tutorial Introduction. In: Srivas, M., Camilleri, A. (eds.) *FMCAD 1996*. LNCS, vol. 1166, pp. 265–269. Springer, Heidelberg (1996)
6. McCasland, R.L., Bundy, A.: MATHsAiD: A Mathematical Theorem Discovery Tool. In: *SYNASC*, pp. 17–22 (2006)
7. Meikle, L.I., Fleuriot, J.D.: Formalizing Hilbert’s Grundlagen in Isabelle/Isar. In: Basin, D., Wolff, B. (eds.) *TPHOLs 2003*. LNCS, vol. 2758, pp. 319–334. Springer, Heidelberg (2003)
8. Milner, R., Bird, R.S.: The Use of Machines to Assist in Rigorous Proof [and Discussion]. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences* 312(1522), 411–422 (1984)

9. Milner, R., Tofte, M., Harper, R., Macqueen, D.: The Definition of Standard ML - Revised. The MIT Press, rev. sub. edn. (May 1997)
10. Moore, E.H.: On the Projective Axioms of Geometry. *Transactions of the American Mathematical Society* 3, 142–158 (1902)
11. Scott, P.: Mechanising Hilbert's Foundations of Geometry in Isabelle. Master's thesis, University of Edinburgh (2008)
12. Scott, P., Fleuriot, J.: Composable Discovery Engines for Interactive Theorem Proving. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 370–375. Springer, Heidelberg (2011)
13. Spivey, J.M.: Algebras for combinatorial search. *Journal of Functional Programming* 19(3-4), 469–487 (2009)
14. Swierstra, S.D., Alcocer, P.R.A., Saraiva, J.: Designing and Implementing Combinator Languages. In: Swierstra, S.D., Oliveira, J.N. (eds.) AFP 1998. LNCS, vol. 1608, pp. 150–206. Springer, Heidelberg (1999)
15. Wadler, P.: Monads for Functional Programming. In: Jeuring, J., Meijer, E. (eds.) AFP 1995. LNCS, vol. 925, pp. 24–52. Springer, Heidelberg (1995)

DynGenPar – A Dynamic Generalized Parser for Common Mathematical Language*

Kevin Kofler and Arnold Neumaier

University of Vienna, Austria
Faculty of Mathematics
Nordbergstr. 15, 1090 Wien, Austria
kevin.kofler@chello.at, Arnold.Neumaier@univie.ac.at

Abstract. This paper introduces a dynamic generalized parser aimed primarily at common natural mathematical language. Our algorithm combines the efficiency of GLR parsing, the dynamic extensibility of tableless approaches and the expressiveness of extended context-free grammars such as parallel multiple context-free grammars (PMCFGs). In particular, it supports efficient dynamic rule additions to the grammar at any moment. The algorithm is designed in a fully incremental way, allowing to resume parsing with additional tokens without restarting the parse process, and can predict possible next tokens. Additionally, we handle constraints on the token following a rule. This allows for grammatically correct English indefinite articles when working with word tokens. It can also represent typical operations for scannerless parsing such as maximal matches when working with character tokens. Our long-term goal is to computerize a large library of existing mathematical knowledge using the new parser, starting from natural language input as found in textbooks or in the papers collected by the digital mathematical library (DML) projects around the world. In this paper, we present the algorithmic ideas behind our approach, give a short overview of the implementation, and present some efficiency results. The new parser is available at <http://www.tigen.org/kevin.kofler/fmathl/dyngenpar/>.

Keywords: dynamic generalized parser, dynamic parser, tableless parser, scannerless parser, parser, parallel multiple context-free grammars, common mathematical language, natural mathematical language, controlled natural language, mathematical knowledge management, formalized mathematics, digital mathematical library.

1 Introduction

The primary target application for our parser is the FMathL (Formal Mathematical Language) project [21]. FMathL is the working title for a modeling and documentation language for mathematics, suited to the habits of mathematicians, to be developed in a project at the University of Vienna. The project

* Support by the Austrian Science Fund (FWF) under contract numbers P20631 and P23554 is gratefully acknowledged.

complements efforts for formalizing mathematics from the Computer Science and automated theorem proving perspective. In the long run, the FMathL system might turn into a user-friendly automatic mathematical assistant for retrieving, editing, and checking mathematics (but also Computer Science and theoretical physics) in both informal, partially formalized, and completely formalized mathematical form.

A major goal of the FMathL project is to build a computer-oriented library of mathematics, in a formalized form the computer can work with, from input as informal as possible. The input language we are working on is a subset of \LaTeX , in which the textual parts are written in a controlled natural language, which we are aiming at growing incrementally to become closer and closer to true natural mathematical language as found in common use (common mathematical language). This is the primary target of our parser. Our current main working grammar is a grammar in extended BNF form [25] produced from German sentences in [20]. We also started work on MathNat [12]. Both those grammars are work in progress; we expect to extend them incrementally over time. Of course, mathematical documents also contain formulas: For those, we will be supporting both sTeX [18] and a subset of the traditional \LaTeX formula syntax as input. Once we have an internal representation of the input, we can not only process it internally, but also generate output, not only in our input formats, but also in domain-specific languages such as, depending on the problem class, AMPL (in which we can already produce some output for simple optimization problems) or the languages used by proof checkers. Our current results are summarized in [24]. We believe our library of mathematical knowledge will nicely supplant DML's human-oriented one. Additionally, our long term goal is to allow computerizing informal knowledge, such as standard textbooks or papers as collected in the DML projects around the world, with minimal user interaction. We expect some interactive input to be needed when processing documents which have not been formalized yet, but the goal is to keep the required interaction as minimal as possible. Finally, we also believe that our software's eventual understanding of mathematics will help building a semantic search engine for DML (and other mathematical) contents.

Our application imposes several design requirements on our parser. It must:

- allow the efficient incremental addition of new rules to the grammar at any time (e.g., when a definition is encountered in a mathematical text – this possibility is typical for mathematical applications –, or to allow teaching the grammar new rules interactively), without recompiling the whole grammar;
- be able to parse more general grammars than just LR(1) or LALR(1) ones – natural language is usually not LR(1), and being able to parse so-called parallel multiple context-free grammars (PMCFGs) [28] is also a necessity for reusing the natural language processing facilities of the Grammatical Framework (GF) [22,23];
- exhaustively produce all possible parse trees (in a packed representation), in order to allow later semantic analysis to select the correct alternative from an ambiguous parse, at least as long as their number is finite;

- support processing text incrementally and predicting the next token (*predictive parsing*);
- be transparent enough to allow formal verification and implementation of error correction in the future;
- support both scanner-driven (for common mathematical language) and scannerless (for some other parsing tasks in our implementation) operation.

These requirements, especially the first one, rule out all efficient parsers currently in use.

We solved this with an algorithm loosely modeled on Generalized LR [29,30], but with an important difference: To decide when to shift a new token and which rule to reduce when, GLR uses complex LR states which are mostly opaque entities in practice, which have to be recomputed completely each time the grammar changes and which can grow very large for natural-language grammars. In contrast, we use the initial graph, which is easy and efficient to update incrementally as new rules are added to the grammar, along with runtime top-down information. The details will be presented in the next section.

This approach allows our algorithm to be both *dynamic*:

- The grammar is not fixed in advance.
- Rules can be added at any moment, even during the parsing process.
- No tables are required. The graph we use instead can be updated very efficiently as rules are added.

and *generalized*:

- The algorithm can parse general PMCFGs.
- For ambiguous grammars, all possible syntax trees are produced.

We expect this parsing algorithm to allow parsing a large subset of common mathematical language and help building a large database of computerized mathematical knowledge. Additionally, we envision potential applications outside of mathematics, e.g., for domain-specific languages for special applications [19]. These are currently mainly handled by scannerless parsing using GLR [31] for context-free grammars (CFGs), but would benefit a lot from our incremental approach. The possibility to add rules at any time, even during parsing, also allows users to quickly add rules if the current grammar does not understand the input, which helps design grammars incrementally. Therefore, we believe that its distinctive features will also make DynGenPar extremely useful on its own, independently of the success of the very ambitious FMathL project.

The algorithm was first presented in our technical report [17].

2 State of the Art

No current parser generator combines all partially conflicting requirements mentioned in the introduction.

Ambiguous grammars are usually handled using **Generalized LR (GLR)** [29,30], needing the compilation of a GLR table, which can take several seconds

or even minutes for large grammars. Such tables can grow extremely large for natural-language grammars. In addition, our parser also supports PMCFGs, whereas GLR only works for context-free grammars (but it may be possible to extend GLR to PMCFGs using our techniques). Costagliola et al. [5] present a predictive parser **XpLR** for visual languages. However, in both cases, since the tables used are mostly opaque, they have to be recomputed completely each time the grammar changes.

The well-known **CYK** algorithm [13,32] needs no tables, but is very inefficient and handles only CFGs. Hinze & Paterson [11] propose a more efficient tableless parser; their idea hasn't been followed up by others.

The most serious competitor to our parser is Angelov's PMCFG parser [3] as found in the code of the **Grammatical Framework** (GF) [22,23], which has some support for natural language and predictive parsing. Alanko and Angelov are currently developing a C version in addition to the existing Haskell implementation. Compared to Angelov's parser, we offer similar features with a radically different approach, which we hope will prove better in the long run. In addition, our implementation already supports features such as incremental addition of PMCFG rules which are essential for our application, which are not implemented in Angelov's current code and which may or may not be easy to add to it. Our parser also supports importing the compiled PGF [4] files from GF, allowing to reuse the rest of the framework. When doing so, as evidenced in section 6, it reaches a comparable performance. Unlike the GF code, we can also enforce next token constraints, e.g., *an restaurant* is not allowed.

3 The DynGenPar Algorithm

In this section, we describe the basics of our algorithm. (Details about the implementation of some features will be presented in section 5.) We start by explaining the design considerations which led to our algorithm. Next, we define the fundamental concept of our algorithm: the initial graph. We then describe the algorithm's fundamental operations and give an example of how they work. Finally, we conclude the section by analyzing the algorithm as a whole.

3.1 Design Considerations

Our design was driven by multiple fundamental considerations. Our first observation was that we wanted to handle left recursion in a most natural way, which has driven us to a bottom-up approach such as LR. In addition, the need for supporting general context-free grammars (and even extensions such as PMCFGs) requires a generalized approach such as GLR. However, our main requirement, i.e., allowing to add rules to the grammar at any time, disqualifies table-driven algorithms such as GLR: recomputing the table is prohibitively expensive, and doing so while the parsing is in progress is usually not possible at all. Therefore, we had to restrict ourselves to information which can be produced dynamically.

3.2 The Initial Graph

To fulfill the above requirements, we designed a data structure we call the *initial graph*. Consider a context-free grammar $G = (N, T, P, S)$, where N is the set of nonterminals, T the set of terminals (tokens), P the set of productions (rules) and S the start symbol. Then the initial graph corresponding to G is a directed labeled multigraph on the set of symbols $\Gamma = N \cup T$ of G , defined by the following criteria:

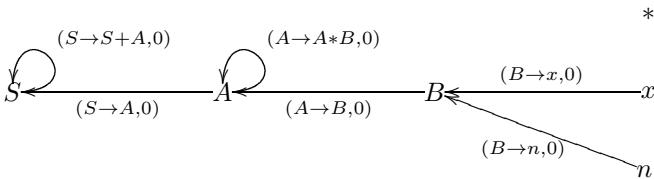
- The tokens T are sources of the graph, i.e., nodes with no incoming edges.
- The graph has an edge from the symbol $s \in \Gamma$ to the nonterminal $n \in N$ if and only if the set of productions P contains a rule $p: n \rightarrow n_1 n_2 \dots n_k s \dots$ with $n_i \in N_0 \forall i$, where $N_0 \subseteq N$ is the set of all those nonterminals from which ε can be derived. The edge is labeled by the pair (p, k) , i.e., the rule (production) p generating the edge and the number k of n_i set to ε .
- In the above, if there are multiple valid (p, k) pairs leading from s to n , we define the edge as a multi-edge with one edge for each pair (p, k) , labeled with that pair (p, k) .

This graph serves as the replacement for precompiled tables and can easily be updated as new rules are added to the grammar.

For example, for the basic expression grammar $G = (N, T, P, S)$ with $N = \{S, A, B\}$, $T = \{+, *, x, n\}$ (where n stands for a constant number, and would in practice have a value, e.g., of `double` type, attached), and P contains the following rules:

- $S \rightarrow S + A \mid A$,
- $A \rightarrow A * B \mid B$,
- $B \rightarrow x \mid n$,

the initial graph would look as follows:



It shall be noted that there is no edge from, e.g., $+$ to S , because $+$ only appears in the middle of the rule $S \rightarrow S + A$ (and the S that precedes it cannot produce the empty string ε), not at the beginning.

To add a new rule, in the simplest case, we just add an edge from the first symbol on the right hand side to the nonterminal on the left hand side. If we have epsilon rules, we may also have to add edges from symbols further on the right to the nonterminal on the left, add the nonterminal on the left to the set of nullable nonterminals if all symbols in the rule can be epsilon, and then go through the rules in the labels of the edges starting at the now nullable nonterminal and recursively repeat the same update process there. But this expensive recursion is rarely needed in practice.

We additionally define *neighborhoods* on the initial graph: Let $s \in \Gamma = N \cup T$ be a symbol and $z \in N$ be a nonterminal (called the *target*). The *neighborhood* $\mathcal{N}(s, z)$ is defined as the set of edges from s to a nonterminal $n \in N$ such that the target z is reachable (in a directed sense) from n in the initial graph. Those neighborhoods can be computed relatively efficiently by a graph walk and can be cached as long as the grammar does not change.

In the example, we would have, e.g., $\mathcal{N}(+, S) = \emptyset$ (because there is no path from $+$ to S), $\mathcal{N}(x, S) = \{B \rightarrow x\}$, and $\mathcal{N}(A, S) = \{S \rightarrow A, A \rightarrow A * B\}$. Note that in the last example, we also have to consider the loop, i.e., the left recursion.

3.3 Operations

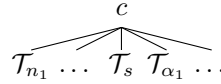
Given these concepts, we define four elementary operations:

- *match_ε(n)*, $n \in N_0$: This operation derives n to ε . It works by top-down recursive expansion, simply ignoring left recursion. This is possible because left-recursive rules which can produce ε necessarily produce infinitely many syntax trees, and we decided to require exhaustive parsing only for a finite number of alternatives.
- *shift*: This operation simply reads in the next token, just as in the LR algorithm.
- *reduce(s, z)*, $s \in \Gamma, z \in N$: This operation reduces the symbol s to the target nonterminal z . It is based on and named after the LR reduce operation, however it operates differently: Whereas LR only reduces a fully matched rule, our algorithm already reduces after the first symbol. This implies that our *reduce* operation must complete the match. It does this using the next operation:
- *match(s)*, $s \in \Gamma = N \cup T$: This operation is the main operation of the algorithm. It matches the symbol s against the input, using the following algorithm:
 1. If $s \in N_0$, try ε -matches first: $m_\varepsilon := match_\varepsilon(s)$. Now we only need to look for nonempty matches.
 2. Start by shifting a token: $t := shift$.
 3. If $s \in T$, we just need to compare s with t . If they match, we return a leaf as our parse tree, otherwise we return no matches at all.
 4. Otherwise (i.e., if $s \in N$), we return $m_\varepsilon \cup reduce(t, s)$.

Given the above operations, the algorithm for *reduce(s, z)* can be summarized as follows:

1. Pick a rule $c \rightarrow n_1 n_2 \dots n_k s \alpha_1 \alpha_2 \dots \alpha_\ell$ in the neighborhood $\mathcal{N}(s, z)$.
2. For each $n_i \in N_0$: $\mathcal{T}_{n_i} := match_\varepsilon(n_i)$.
3. s was already recognized, let \mathcal{T}_s be its syntax tree.
4. For each $\alpha_j \in \Gamma = N \cup T$: $\mathcal{T}_{\alpha_j} := match(\alpha_j)$. Note that this is a top-down step, but that the *match* operation will again do a bottom-up shift-reduce step.

5. The resulting syntax tree is:



6. If $c \neq z$, continue reducing recursively ($reduce(c, z)$) until the target z is reached. We also need to consider $reduce(z, z)$ to support left recursion; this is the only place in our algorithm where we need to accomodate specifically for left recursion.

If we have a conflict between multiple possible *reduce* operations, we need to consider all the possibilities. We then unify our matched parse trees into DAGs wherever possible to both reduce storage requirements and prevent duplicating work in the recursive *reduce* steps. This is described in more detail in section 5.

Our algorithm is initialized by calling $match(S)$ on the start symbol S of the grammar. The rest conceptually happens recursively. The exact sequence of events in our practical implementation, which allows for predictive parsing, is described in section 5.

3.4 Example

As an example, we show how our algorithm works on the basic expression grammar from section 3.2. The example was chosen to be didactically useful rather than realistic: In practice, we work with grammars significantly more complex than this example. It shall be noted that in this example, the set N_0 of nonterminal which can be derived to ε is empty. Handling ε -productions requires some technical tricks (skipped initial nonterminals with empty derivation in rules, $match_\varepsilon$ steps), but does not impact the fundamental algorithm.

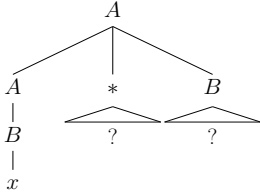
We consider the input $x * x$, a valid sentence in the example grammar. We will denote the cursor position by a dot, so the initial input is $.x * x$. The algorithm always starts by matching the start category, thus the initial step is $match(S)$. The *match* step starts by shifting a token, then tries to reduce it to the symbol being matched. In this case, the *shift* step produces the token x , the input is now $x.* x$, and the next step is $reduce(x, S)$, after which the parsing is complete.

It is now the *reduce* task’s job to get from x to S , and to complete the required rules by shifting and matching additional tokens. To do this, it starts by looking for a way to get closer towards S , by looking at the neighborhood $\mathcal{N}(x, S) = \{B \rightarrow x\}$. In this case, there is only one rule in the neighborhood, so we reduce that rule. The right hand side of the rule is just x , so the rule is already completely matched, there are no symbols left to match. We remember the parse tree $B - x$ and proceed recursively with $reduce(B, S)$.

Now we have $\mathcal{N}(B, S) = \{A \rightarrow B\}$. Again, there is only a single rule that matches and it is fully matched, so we reduce it, remember the parse tree $A - B - x$ and continue the recursion with $reduce(A, S)$.

This time, the neighborhood $\mathcal{N}(A, S) = \{S \rightarrow A, A \rightarrow A * B\}$ contains more than one matching rule, we have a *reduce-reduce conflict*. Therefore, we have to consider both possibilities, as in GLR. If we attempt to reduce $S \rightarrow A$, the

parsing terminates here (or we try reducing the left-recursive $S \rightarrow S + A$ rule and hit an error on the unmatched $+$ token), but the input is not consumed yet, thus we hit an error. Therefore, we retain only the option of reducing the left-recursive $A \rightarrow A * B$ rule. This time, there are two remaining tokens: $*$ and B , thus we proceed with $match(*)$ and $match(B)$. Our parse tree matched so far is



The $match(*)$ operation is trivial: $*$ is a token, so we only need to *shift* the next token and compare it to $*$. The input is now $x * .x$, and the $match(B)$ step proceeds by a last *shift* consuming the last token, and a $reduce(x, B)$ which is also trivial because $\mathcal{N}(x, B) = \{B \rightarrow x\}$.

Thus the reduction of the left-recursive rule $A \rightarrow A * B$ is complete and we recursively proceed with another $reduce(A, S)$. This time, attempting to reduce the left-recursive rule again yields an error (there is no input left to match the $*$ against) and we reduce $S \rightarrow A$, giving the final parse tree.

A similar, but slightly longer example can be found in the slides [16].

3.5 Analysis

The above algorithm combines enough bottom-up techniques to avoid trouble with left recursion with sufficient top-down operation to avoid the need for tables while keeping efficiency. The initial graph ensures that the bottom-up steps never try to reduce unreachable rules, which is the main inefficiency in existing tableless bottom-up algorithms such as CYK [13][32].

One disadvantage of our algorithm is that it produces more conflicts than LR or GLR, for two reasons: Not only are we not able to make use of any lookahead tokens, unlike common LR implementations, which are LR(1) rather than LR(0), but we also already have to reduce after the first symbol, whereas LR only needs to make this decision at the end of the rule. However, this drawback is more than compensated by the fact that we need no states nor tables, only the initial graph which can be dynamically updated, which allows dynamic rule changes. In addition, conflicts are not fatal because our algorithm is exhaustive (like GLR), and we designed our implementation to keep its efficiency even in the presence of conflicts; in particular, we never execute the same $match$ step at the same text position more than once.

4 Implementation

In this section, we first give an overview of the technologies and the license chosen for our implementation. Then, we describe how it integrates into our main application software.

The DynGenPar implementation is available for free download at [15].

4.1 Technologies and Licensing

Our implementation is written in C++ using the Qt [1] toolkit. It is licensed under the GNU General Public License [9,10], version 2 or later.

We also implemented Java bindings using the Qt Jambi [2] binding generator to allow its usage in Java programs.

4.2 Integration into FMathL Concise

The Java bindings are used in the Concise [27] GUI of the FMathL project [21]. Concise is a framework for viewing and manipulating, both graphically and programmatically, semantic graphs. It is the main piece of software in our application. Concise offers editable views of semantic content in the form of graphs, records or text, can execute programs operating on that content, and supports importing information from and exporting it to various types of files. It is written in Java.

The Concise GUI fully integrates our DynGenPar parser into our application's workflow. The FMathL type system [26] is represented in the form of text files called *type sheets*. Those type sheets can not only represent a pure type hierarchy, but also carry grammatical annotations, which allow the type system to double as a grammar. Concise can import such type sheets at runtime and automatically convert them to grammar rules suitable for DynGenPar. It can then parse documents using the converted grammar.

This feature allows to read user-written rules into the parser at runtime, rather than hardcoding them as C++ code or compiling them with the Grammatical Framework (GF) to its binary PGF format. Concise type sheets represent a user-friendly mechanism for specifying rules which can be easily converted to our internal representation. The feature is thus an ideal showcase for the dynamic properties of our algorithm.

5 Implementation Considerations

This section documents some tweaks we made to the above basic algorithm to improve efficiency and provide additional desired features. We describe the modifications required to support predictive parsing, efficient exhaustive parsing, peculiarities of natural language, arbitrary rule labels, custom parse actions and next token constraints. Next, we briefly introduce our flexible approach to lexing. Finally, we give a short overview on interoperability with the Grammatical Framework (GF).

5.1 Predictive Parsing

The most intuitive approach to implement the above algorithm would be to use straight recursion with implicit parse stacks and backtracking. However, that approach does not allow incremental operation, and it does not allow discarding

short matches (i.e., prefixes of the input which already match the start symbol) until the very end. Therefore, we replaced the backtracking by explicit parse stacks, with token shift operations driving the parse process: Each time a token has to be shifted, the current stack is saved and processing stops there. Once the token is actually shifted, all the pending stacks are processed, with the shift executed. If there is no valid match, the parse stack is discarded, otherwise it is updated. We also remember complete matches (where the entire starting symbol S was matched) and return them if the end of input was reached, otherwise we discard them when the next token is shifted. This method allows for incremental processing of input and easy pinpointing of error locations. It also allows changing the grammar rules for a specific range of text only.

The possible options for the next token and the nonterminal generating it can be predicted. This is implemented in a straightforward way by inspecting the parse stacks for the next pending match, which yields the next highest-level symbol, and if that symbol is a nonterminal, performing a top-down expansion (ignoring left recursion) on that symbol to obtain the possible initial tokens for that symbol, along with the nonterminal directly producing them. Once a token is selected, parsing can be continued directly from where it was left off using the incremental parsing technique described in the previous paragraph.

5.2 Efficient Exhaustive Parsing

In order to achieve efficiency in the presence of ambiguities, the parse stacks are organized in a DAG structure similar to the GLR algorithm's graph-structured stacks. [29,30] In particular, a *match* operation can have multiple parents, and our algorithm produces a unified stack entry for identical match operations at the same position, with all the parents grouped together. This prevents having to repeat the match more than once. Only once the match is completed, the stacks are separated again.

Parse trees are represented as packed forests. Top-down sharing is explicit: Any node in a parse tree can have multiple alternative subtrees, allowing to duplicate only the local areas where there are ambiguities and share the rest. This representation is created by explicit unification steps. This sharing also ensures that the subsequent *reduce* operations will be executed only once on the shared parse DAG, not once per alternative. Bottom-up sharing, i.e., multiple alternatives having common subtrees, is handled implicitly through the use of reference-counted implicitly shared data structures, and through the graph-structured stacks ensuring that the structures are parsed only once and that the same structures are referenced everywhere.

5.3 Rule Labels

Our implementation allows labeling rules with arbitrary data. The labels are reproduced in the parse trees. This feature is essential in many applications to efficiently identify the rule which was used to derive the relevant portion of the parse tree.

5.4 Custom Parse Actions

The algorithm as described in section 3 generates only a plain parse tree and cannot execute any other actions according to the grammar rules. But in order to efficiently support things such as mathematical definitions, we need to be able to automatically trigger the addition of a new grammar rule (which can be done very efficiently by updating the initial graph) by the encountering of the definition. Therefore, the implementation makes it possible to attach an action to a rule, which will be executed when the rule is matched. This is implemented by calling the action at the end of a *matchRemaining* step, when the full rule has been matched.

5.5 Token Sources

The implementation can be interfaced with several different types of token sources, e.g., a Flex 7 lexer, a custom lexer, a buffer of pre-lexed tokens, a dummy lexer returning each character individually etc. The token source may or may not attach data to the tokens, e.g., a lexer will want to attach the value of the integer to `INTEGER` tokens.

The token source can also return a whole parse tree instead of the usual leaf node. That parse tree will be attached in place of the leaf. This feature makes hierarchical parsing possible: Using this approach, the token source can run another instance of the parser (DynGenPar is fully reentrant) or a different parser (e.g., a formula parser) on a token and return the resulting parse tree.

5.6 Natural Language

Natural language, even the subset used for mathematics, poses some additional challenges to our implementation. There are two ways in which natural language is not context free: *attributes* (which have to agree, e.g., for declination or conjugation) and other context sensitivities best represented by PMCFGs 28.

Agreement issues are the most obvious context sensitivity in natural languages. However, they are easily addressed: One can allow each nonterminal to have *attributes* (e.g., the grammatical number, i.e., singular or plural), which can be *inherent* to the grammatical category (e.g., the number of a noun phrase) or variable *parameters* (e.g., the number for a verb). Those attributes must *agree*, which in practice means that each attribute must be inherent for exactly one category and that the parameters inherit the value of the inherent attribute. While this does not look context-free at first, it can be transformed to a CFG (as long as the attribute sets are finite) by making a copy of a given nonterminal for each value of each parameter and by making a copy of a given production for each value of each inherent attribute used in the rule. This transformation can be done automatically, e.g., the GF compiler does this for grammars written in the GF programming language.

A less obvious, but more difficult problem is given by split categories, e.g., verb forms with an auxiliary and a participle, which grammatically belong together, but are separated in the text. The best solution in that case is to generalize the concept of CFGs to PMCFGs [28], which allow nonterminals to have multiple dimensions. Rules in a PMCFG are described by functions which can use the same argument more than once, in particular also multiple elements of a multi-dimensional category. PMCFGs are more expressive than CFGs, which implies that they cannot be transformed to CFGs. They can, however, be parsed by context-free approximation with additional constraints. Our approach to handling PMCFGs is based on this idea. However, we do not use the naive and inefficient approach of first parsing the context-free approximation and then filtering the result, but we enforce the constraints directly during parsing, leading to maximum efficiency and avoiding the need for subsequent filtering. This is achieved by keeping track of the constraints that apply, and immediately expanding rules in a top-down fashion (during the *match* step) if a constraint forces the application of a specific rule. The produced parse trees are CFG parse trees which are transformed to PMCFG syntax trees by a subsequent unification algorithm, but the parsing algorithm ensures that only CFG parse trees which can be successfully unified are produced, saving time both during parsing and during unification. This unification process uses DynGenPar’s feature to attach, to CFG rules, arbitrary rule labels which will be reproduced in the parse tree: The automatically generated label of the CFG rule is an object containing a pointer to the PMCFG rule and all other information needed for the unification.

5.7 Next Token Constraints

Our implementation also makes it possible to attach constraints on the token following a rule, i.e., that said token must or must not match a given context-free symbol, to that rule. We call such constraints *next token constraints*. This feature can be used to implement scannerless parsing patterns, in particular, maximally-matched character sequences, but also to restrict the words following e.g., “a” or “an” in word-oriented grammars. We implement this by collecting the next token constraints as rules are reduced or expanded and attaching them to the parse stacks used for predictive parsing. Each time a token is shifted, before processing the pending stacks, we check whether the shifted token fulfills the pending constraints and reject the stacks whose constraints aren’t satisfied.

5.8 Interoperability with GF

Our implementation can import PGF [4] grammar files produced by the Grammatical Framework (GF) [22,23], a binary format based on PMCFGs. This is handled by converting them to PMCFG standard form, with a few extensions supported by our parser:

- Additional context-free rules can be given, the left-hand sides of which can be used as “tokens” in the expression of PMCFG functions.

- Next token constraints can be used. This and the previous extension are required to support GF’s rules for selecting e.g., “a” vs. “an”.
- PMCFG functions can be given a token (or a context-free nonterminal as above) as a parameter, in which case the syntax tree will reproduce the parse tree of that symbol verbatim, including attached data, if any. This extension is required to support GF’s builtin `String`, `Int` and `Float` types.

We also implemented a GF-compatible lexer.

6 Results

Our main achievement is the dynamism of the algorithm. However, the algorithm must also be fast enough for practical use (in particular, faster than recompiling the grammar with a static parser). Therefore, we compared the speed of our implementation to the well-established GNU Bison [8] parser on a hierarchical (two-layer) grammar we devised for the Naproche [14,6] language: There are 2 context-free grammars, one for text and one for formulas, each using a lexer based on Flex [7]. In one version of our Naproche parser, the 2 context-free grammars are processed with Bison (using its support for GLR parsing), in the other with DynGenPar. We measured the times required to compile the code to an executable (using GCC with `-O2` optimization), to convert the grammar rules to the internal representation (GLR tables for Bison, initial graphs for DynGenPar), and to actually parse a sample input (representing the Burali-Forti paradoxon in Naproche). It shall be noted that for Bison, the grammar conversion is done before the compilation, so the compilation time also has to be considered when working with dynamically changing grammars, whereas DynGenPar can convert grammars at runtime. Our test machine is a notebook with a Core 2 Duo T7700 (2×2.40 GHz) and 4 GiB RAM running Fedora 16 x86_64. For each measurement, we averaged the execution times of 100 tests (except for the compilation time of DynGenPar, where we used only 3 tests due to time constraints) and took the median of 3 attempts. Our results are summarized in table 1.

We conclude that, while Bison is 4 to 5 times faster at pure parsing, DynGenPar is much faster at adapting to changed grammars. The time required to compile modified grammars makes Bison entirely unsuitable for applications where the grammar can change dynamically. Even if Bison were changed to allow loading a different LR table at runtime, it would still take 11 times longer than DynGenPar to process our fairly small two-layered grammar, and we expect the

Table 1. Benchmarking results on the Naproche grammar

	compilation time	grammar conversion time	parsing time (Burali-Forti)
Bison	2722 ms	83 ms*	2.79 ms
DynGenPar	20890 ms	7.54 ms	12.2 ms**

* ... at compile time, thus requires recompilation

** ... total execution time of 19.7 ms minus grammar conversion time

discrepancy to only grow as the grammar sizes increase. (Moreover, DynGenPar can handle dynamic rule addition, so in many cases even the 7.54 ms for grammar conversion can be saved.) In the worst case, where we have a new input for an existing grammar and do not have the initial graph in memory, DynGenPar (19.7 ms) is still only 7 times slower than Bison (2.79 ms), even though the latter was optimized specifically for this usecase and DynGenPar was not.

We also benchmarked our support for PGF [4] grammar files produced by the Grammatical Framework (GF) [22,23] against two PGF runtimes provided by the GF project (we used a snapshot of the repository from February 24): the production runtime written in Haskell and the new experimental runtime written in C. As an example grammar, we used GF’s *Phrasebook* example, which is the one explicitly documented as being supported by the current version of GF’s C runtime, with the sample sentence *See you in the best Italian restaurant tomorrow!*, a valid sentence in the *Phrasebook* grammar. (We also tried parsing with the full English resource grammar, but DynGenPar would not scale to such huge grammars and did not terminate in a reasonable time.) We measured the time to produce the syntax tree only, without outputting it. The tests were run on the same Core 2 Duo notebook as above. Again, for each measurement, we averaged the execution times of 100 tests and took the median of 3 attempts. Our results are summarized in table 2.

We conclude that DynGenPar is competitive in speed with both GF runtimes on practical application grammars. In addition, both GF runtimes happily accept the incorrect input *Where is an restaurant?* (should be *a restaurant*), whereas DynGenPar can enforce the next token constraint.

Table 2. Benchmarking results on the GF *Phrasebook* grammar

	parsing time (<i>See you in the best Italian restaurant tomorrow!</i>)
GF Haskell runtime	37 ms
GF C runtime	84 ms
DynGenPar	81 ms

7 Conclusion

We introduced DynGenPar, a dynamic generalized parser for common mathematical language, presented its requirements, the basics of the algorithm and the tweaks required for an efficient implementation, and compared our approach with the state of the art, evidencing the huge advancements we made.

However, there is still room for even more features, which will bring us further towards our goal of computerizing a library of mathematical knowledge:

- context-sensitive constraints on rules: Currently, we support only some very specific types of context-sensitive constraints, i.e., PMCFG and next token constraints. We would like to support more general types of constraints, and our algorithm is designed to accomodate that. The main research objective here will be to figure out the class of constraints that is actually needed.

- stateful parse actions: Custom parse actions currently have access only to minimal state information. We plan to make more state available to parse actions to provide as much flexibility as we find will be needed.
- a runtime parser for rules: Reading rules into the parser from a user-writable format at runtime, rather than from precompiled formats such as machine code or PGF grammars, is currently possible through the Concise [27] GUI. We are considering implementing a mechanism for specifying rules at runtime within DynGenPar. However, this has low priority for us because we use the mechanism provided by Concise in our application.
- scalability to larger PMCFGs: Currently, we have several optimizations which improve scalability, but only apply in the context-free case. In order to be able to process huge PMCFGs such as the resource grammars of the Grammatical Framework, we need to find ways to improve scalability also in the presence of constraints.
- error correction: At this time, DynGenPar only has basic error detection and reporting: A parse error happens when a shifted token is invalid for all pending parse stacks. We would like to design intelligent ways to actually correct the errors, or suggest corrections to the user. This is a long-term research goal.

Our hope is that the above features will make it easy to parse enough mathematical text to build a large database of mathematical knowledge, as well as adapting to a huge variety of applications in mathematics and beyond.

References

1. Qt – Cross-platform application and UI framework, <http://qt.nokia.com>
2. Qt Jambi – The Qt library for Java, <http://qt-jambi.org>
3. Angelov, K.: Incremental parsing with parallel multiple context-free grammars. In: Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics, pp. 69–76 (2009)
4. Angelov, K., Bringert, B., Ranta, A.: PGF: A Portable Run-Time Format for Type-Theoretical Grammars. *Journal of Logic, Language and Information* 19(2), 201–228 (2010)
5. Costagliola, G., Deufemia, V., Polese, G.: Visual language implementation through standard compiler-compiler techniques. *Journal of Visual Languages & Computing* 18(2), 165–226 (2007); selected papers from Visual Languages and Computing 2005
6. Cramer, M., Fisseni, B., Koepke, P., Kühlwein, D., Schröder, B., Veldman, J.: The Naproche Project Controlled Natural Language Proof Checking of Mathematical Texts. In: Fuchs, N.E. (ed.) CNL 2009. LNCS, vol. 5972, pp. 170–186. Springer, Heidelberg (2010)
7. Flex Project: flex: The Fast Lexical Analyzer, <http://flex.sourceforge.net>
8. Free Software Foundation: Bison – GNU parser generator, <http://www.gnu.org/software/bison>
9. Free Software Foundation: GNU General Public License (GPL) v2.0 (June 1991), <http://www.gnu.org/licenses/old-licenses/gpl-2.0>
10. Free Software Foundation: GNU General Public License (GPL) v3.0 (June 2007), <http://www.gnu.org/licenses/gpl-3.0>

11. Hinze, R., Paterson, R.: Derivation of a typed functional LR parser (2003)
12. Humayoun, M.: Developing the System MathNat for Automatic Formalization of Mathematical texts. Ph.D. thesis, University of Grenoble (2012), <http://www.lama.univ-savoie.fr/~humayoun/phd/mathnat.html>
13. Kasami, T.: An efficient recognition and syntax analysis algorithm for context-free languages. Tech. Rep. AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA (1965)
14. Koepke, P., Schröder, B., Buechel, G., et al.: Naproche – Natural language proof checking, <http://www.naproche.net>
15. Kofler, K.: DynGenPar – Dynamic Generalized Parser, <http://www.tigen.org/kevin.kofler/fmathl/dyngenpar>
16. Kofler, K., Neumaier, A.: The DynGenPar Algorithm on an Example, slides, <http://www.tigen.org/kevin.kofler/fmathl/dyngenpar-example.pdf>
17. Kofler, K., Neumaier, A.: A Dynamic Generalized Parser for Common Mathematical Language. In: Work-in-Progress Proceedings of CICM/MKM (2011), <http://www.tigen.org/kevin.kofler/fmathl/dyngenpar-wip.pdf>
18. Kohlhase, M.: Using LaTeX as a Semantic Markup Format. Mathematics in Computer Science 2.2, 279–304 (2008)
19. Mernik, M., Heering, J., Sloane, A.: When and how to develop domain-specific languages. ACM Computing Surveys (CSUR) 37(4), 316–344 (2005)
20. Neumaier, A.: Analysis und lineare Algebra, unpublished lecture notes, <http://www.mat.univie.ac.at/~neum/FMathL/ALA.pdf>
21. Neumaier, A.: FMathL – Formal Mathematical Language, <http://www.mat.univie.ac.at/~neum/fmathl.html>
22. Ranta, A.: Grammatical Framework: A Type-Theoretical Grammar Formalism. Journal of Functional Programming 14(2), 145–189 (2004)
23. Ranta, A., Angelov, K., Hallgren, T., et al.: GF – Grammatical Framework, <http://www.grammaticalframework.org>
24. Schodl, P.: Foundations for a Self-Reflective, Context-Aware Semantic Representation of Mathematical Specifications. Ph.D. thesis, University of Vienna (2011), http://www.mat.univie.ac.at/~schodl/pdfs/diss_online.pdf
25. Schodl, P., Neumaier, A.: An experimental grammar for German mathematical text. Tech. rep., University of Vienna (2009), <http://www.mat.univie.ac.at/~neum/FMathL/ALA-grammar.pdf>
26. Schodl, P., Neumaier, A.: The FMathL type system. Tech. rep., University of Vienna (2011), <http://www.mat.univie.ac.at/~neum/FMathL/types.pdf>
27. Schodl, P., Neumaier, A., Kofler, K., Domes, F., Schichl, H.: Towards a Self-reflective, Context-aware Semantic Representation of Mathematical Specifications. In: Kallrath, J. (ed.) Algebraic Modeling Systems – Modeling and Solving Real World Optimization Problems, ch. 2. Springer (2012)
28. Seki, H., Matsumura, T., Fujii, M., Kasami, T.: On multiple context-free grammars. Theoretical Computer Science 88(2), 191–229 (1991)
29. Tomita, M.: An Efficient Augmented Context-Free Parsing Algorithm. Computational Linguistics 13(1-2), 31–46 (1987)
30. Tomita, M., Ng, S.: The Generalized LR parsing algorithm. In: Tomita, M. (ed.) Generalized LR Parsing, pp. 1–16. Kluwer (1991)
31. Visser, E.: Scannerless generalized-LR parsing. Tech. Rep. P9707, Programming Research Group, University of Amsterdam (1997)
32. Younger, D.: Recognition and parsing of context-free languages in time n^3 . Information and Control 10(2), 189–208 (1967)

Writing on Clouds

Vadim Mazalov and Stephen M. Watt

Department of Computer Science
The University of Western Ontario
London Ontario, Canada N6A 5B7
{vmazalov, Stephen.Watt}@uwo.ca

Abstract. While writer-independent handwriting recognition systems are now achieving good recognition rates, writer-dependent systems will always do better. We expect this difference in performance to be even larger for certain applications, such as mathematical handwriting recognition, with large symbol sets, symbols that are often poorly written, and no fixed dictionary. In the past, to use writer-dependent recognition software, a writer would train the system on a particular computing device without too much inconvenience. Today, however, each user will typically have multiple devices used in different settings, or even simultaneously. We present an architecture to share training data among devices and, as a side benefit, to collect writer corrections over time to improve personal writing recognition. This is done with the aid of a handwriting profile server to which various handwriting applications connect, reference, and update. The user's handwriting profile consists of a cloud of sample points, each representing one character in a functional basis. This provides compact storage on the server, rapid recognition on the client, and support for handwriting neatening. This work uses the word “cloud” in two senses. First, it is used in the sense of cloud storage for information to be shared across several devices. Secondly, it is used to mean clouds of handwriting sample points in the function space representing curve traces. We “write on clouds” in both these senses.

Keywords: Handwriting Recognition, Mathematical Handwriting Recognition, Cloud Computing, Service Oriented Architecture.

1 Introduction

We are interested in online recognition of handwritten mathematics. The widespread use of hand-held mobile devices and tablets has created a ubiquitous environment for two-dimensional math input. Writing mathematics on a digital canvas is similar to traditional pen-on-paper input. It does not require learning any typesetting languages and can be efficient, given a robust and reliable implementation. According to one study [1], pen-based input of mathematics is about three times faster and two times less error-prone than standard keyboard- and mouse-driven techniques. However, recognition of mathematics is a harder problem than recognition of natural language text.

In our classification paradigm, a character is represented by the coefficients of an approximation of trace curves with orthogonal polynomials [4]. Classification is based on computation of the distance to convex hulls of nearest neighbours in the space of coefficients of approximation. Typically, the method does not require many training samples to discriminate a class. However, because there are a large number of classes in handwritten mathematics, the training dataset may contain tens of thousands of characters. The underlying recognition model allows the dataset to evolve over the course of normal use. Furthermore, as a user makes corrections to mis-recognized input, new training data is obtained. Therefore, synchronization of the dataset across several pen-based devices may become tiresome. To address this aspect, we propose to delegate the storage of the training database, as well as some of the recognition tasks to a cloud.

In the present work we describe a cloud-based recognition architecture. It has potential to be beneficial not only to end users, but also to researchers in the field. A cloud infrastructure can assist in the capture of recognition history. The “knowledge” obtained from the public usage of the recognition software can help to improve the accuracy continuously. This serves as a basis for an adaptive recognition that results in asymptotic increase of user-, region-, or country-centered classification rate. Additionally, such a model has a number of other advantages: First, it allows the writer to train the model only once and then use the cloud with any device connected to the Internet. Secondly, it gives the user access to various default collections of training samples across different alphabets (e.g. Cyrillic, Greek, Latin), languages (e.g. English, French, Russian), and domains (e.g. regular text, mathematics, musical notation, chemical formulae). Thirdly, it provides a higher level of control over the classification results and correction history.

The architecture we present may be applicable to a variety of recognition methods across different applications, including voice recognition, document analysis, or computer vision. To demonstrate its use in recognizing handwritten mathematical characters, we have performed an experiment to measure the error convergence as a function of the input size and find an average number of personal samples in a class to achieve high accuracy.

The rest of the paper is organized as follows. Section 2 introduces the character approximation and recognition foundation, as well as some preliminary concepts required by the proposed architecture. Section 3 describes the cloud-based recognition framework, starting by giving an overview of the components. Then the flow of recognition and correction, as well as possible manipulations of clusters, are presented. Section 4 describes details of the implementation of the system, the structure of a personal profile, the interface for training and recognition, the server side, as well as calligraphic representation of recognized characters. The recognition error decrease as a function of the input size of a writer is presented in Section 5. Section 6 concludes the paper.

2 Preliminaries and Related Work

2.1 Recognition Aspects

In an online classification environment, a curve may be given as an ordered set of points in a Euclidean plane. Devices are capable of sampling the coordinates of a stylus as functions of time. The inputs to online classification are typically given as vectors of pen coordinates, represented as real numbers received at a fixed frequency [4]. In addition, some devices can collect other information, such as pressure or pen angle, as well as spatial coordinates when a stylus is not touching the surface. We however do not rely on this additional information so we can maintain hardware independence.

The input traces may be regarded as parametric functions and we may represent these using standard approximation methods as truncated orthogonal polynomial series

$$X(t) \approx \sum_{i=0}^d x_i B_i(\lambda), \quad Y(\lambda) \approx \sum_{i=0}^d y_i B_i(\lambda)$$

where $B_i(\lambda)$ are the orthogonal basis polynomials, e.g. Chebyshev, Legendre or Legendre-Sobolev polynomials, and λ is a parameter, e.g. time or arc-length [4]. Multi-stroke characters may be represented by concatenating the coordinate sequences of the strokes.

Having the coefficients of approximation, a character can be represented as the tuple

$$x_0, x_1, \dots, x_d, y_0, y_1, \dots, y_d.$$

In this vector, coefficients x_0 and y_0 give the initial position of the sample and can be neglected to normalize location of the character. Dividing the rest of the vector by the Euclidean norm will normalize the sample with respect to size. We base classification on a distance (in some norm) to the convex hull of k nearest neighbours in the space of coefficients [4].

2.2 Architectural Aspects

Cloud computing allows remote, distributed storage and execution. The economic stimuli for providing software services in a cloud infrastructure are similar to those for centralized supply of water or electricity. This relieves consumers from a number of issues associated with software maintenance, while the provider may continuously improve the service.

Agility of a cloud service is usually achieved by its internal organization according to the principles of the Service-Oriented Architecture (SOA). SOA allows splitting computational tasks into loosely coupled units, services, that can be used in multiple unassociated software packages. An external application executes a service by making a call through the network. The service consumer remains independent of the platform of the service provider and the technology with which the service was developed.

2.3 Related Work

Several related projects have been described that mostly target development of managed experimental repositories and resource sharing in the context of: document analysis [7], astronomical observations [14], or environmental research [2]. In contrast, our primary objective is improvement of usability of recognition software across different pen-based devices. Collecting a comprehensive database that facilitates research is the second priority.

3 Clouds Serving Clouds

Touch screens with the ability to handle digital ink are becoming *de facto* standards of smart phones and tablet computers. The variety of such platforms challenges conventional recognition applications because:

- Certain mobile devices have limited storage capacity and computational power, restricting ink storage and processing. Recognition of handwritten math requires extra resources to build classification theories and to calculate the confidence of each theory [3].
- Development of a single recognition engine that runs efficiently across all the platforms is not easy, and in most cases a trade off has to be made, affecting classification performance.
- The evolving personal training datasets and correction histories are not synchronized across the devices.

Similar to the software as a service delivery model, we propose to have digital ink collected and, possibly, processed through a thin client, but its storage and some computationally intensive procedures are performed centrally in the cloud.

From the high-level, the system contains the following elements

- *Canvas* of a pen-based device, that can collect digital ink.
- *HLR* (High-Level Recognizer) accepts raw ink from the canvas and performs initial preprocessing of the ink.
- *Recognizer* is a character recognition engine, developed according to the principles described in [4].
- *Database* stores personal handwriting data, profiles of samples, correction history, etc.

Profiles of training samples are clouds of points in the space of approximated curves, each point being one character. These points are saved in a database in the cloud. When users sign up for the service, they are assigned a default dataset of training samples. If a person has several handwriting domains (e.g. different fields using mathematics, physics, music, etc), each domain should have a separate dataset, and the recognition application should allow switching between the subjects. The user shapes the datasets through a series of recognitions and corrections. Below, we show experimentally that the number of corrections decreases over time and eventually becomes quite small.

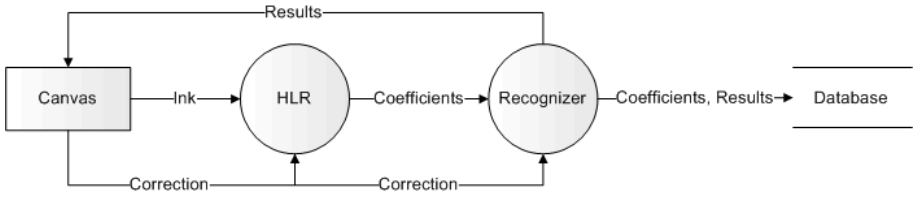


Fig. 1. The data flow diagram for recognition and correction

3.1 Recognition Flow

The overall recognition flow is shown in Figure 1. The High-Level Recognizer (HLR) accepts raw ink from the canvas and preprocesses it. The output of the HLR is available to the recognizer in the form of normalized coefficients. The coefficients are recognized. The results of classification are sent to the canvas and saved in the database.

Representation of Characters. For a single-stroke character, after approximation of coordinates with truncated orthogonal series, the sample can be represented as

$$\frac{1}{\|x, y\|}, x_0, y_0, x'_1, y'_1, \dots, x'_d, y'_d \tag{1}$$

where x_0, y_0 are Legendre-Sobolev coefficients that control the initial position of the character, $x'_1, y'_1, \dots, x'_d, y'_d$ are normalized coefficients, and $\|x, y\|$ is the Euclidean norm of the vector [4]

$$x_1, \dots, x_d, y_1, \dots, y_d$$

The first three elements in (1) are ignored during recognition, but used in restoring the initial size and location of the character.

For a multi-stroke symbol, coefficients are computed for every stroke, as described for a single-stroke character, and also for all strokes joined sequentially. Coefficients of strokes are used for display of the sample and normalized coefficients of joined strokes are used for classification.

The described representation of samples allows significant saving on storage space and computations, since coefficients of symbols can be directly used in recognition without repetitive approximation [10]. However, this compression scheme is lossy and should not be used when precision of digital ink is of high importance, e.g. in applications that involve processing of personal signatures.

Recognition. Individual handwriting can differ significantly from the default collection of training samples. This is illustrated by the historical use of a personal signature as a form of authentication of documents. It is to be expected that a successful recognition system should adapt to personal writing style. With

$$\text{coefficients} ::= \frac{1}{\|x, y\|}; x_0; y_0; x'_1; y'_1; \dots; x'_d; y'_d$$

```

msg ::= <m:Process>
      <m:mt>coefficients</m:mt>
      ( <m:tr>coefficients</m:tr> <m:tr>coefficients</m:tr> + )?
</m:Process>

```

Fig. 2. The format of the SOAP message sent to the cloud

```

...
<soap:Body xmlns:m="http://www.inkml.org/processing">
  <m:Process>
    <m:mt>0.005;94;-91;11;2;-14;64;-70;
      -18;1;-75;14;14;8;4;-2;4;0;-9;5;10;-11;5;</m:mt>
  </m:Process>
</soap:Body>
...

```

Listing 1. An example of the body of a SOAP message for a single-stroke character

k -nearest neighbors and related methods, the test sample can be easily introduced to the training set after classification. This facilitates adaptive recognition, since the model remains synchronized with the writer's style.

Two modes of recognition are possible, *local* and *remote*.

Local recognition is suitable for devices with sufficient computational capabilities. In this mode, the points that form the convex hulls of classes are stored on the device locally and periodically synchronized with the server. Synchronization can be performed through a profile of samples. The local recognition mode is useful when the user does not have a network connection and therefore can not take advantage of the remote recognition described below.

In *remote recognition* mode, digital curves are collected and preprocessed locally, and the coefficients are sent to a remote recognition engine. Having recognized the character, the server returns encoding of the symbol and nearest candidates. This mode allows to minimize the load on the bandwidth, since the training dataset does not have to be synchronized with the device.

Coefficients can be transmitted in the body of a SOAP message, using the syntax shown in Figure 2. The element `<m:mt>` contains the normalization weight, the original coefficients of the 0-degree polynomials, and the normalized coefficients used in recognition. Additionally, for a multi-stroke sample, the `<m:tr>` element is used to represent each stroke independently. Examples of messages for a single-stroke and a multi-stroke character are shown in Listing 1 and Listing 2 respectively. The bodies of the SOAP messages contain enough information for both recognition and restoring approximate representation of a character in its initial position.

The results of recognition can be returned in a SOAP message, as shown in Listing 3. The body contains Unicode values of the top candidates to enable the client application to visualize recognized characters in a printed format. For calligraphic rendering, corresponding coefficients can be included as well.

```

...
<soap:Body xmlns:m="http://www.inkml.org/processing">
  <m:Process>
    <m:mt>1;0;0;-5;-22;-14;-15;-44;-72;20;13;-27;43;4;
      -28;48;-1;-10;16;-32;-17;-1;-12;</m:mt>
    <m:tr>0.005;92;-85;-1;3;-7;62;-79;-30;
      4;-61;32;4;-2;15;-4;-4;6;-3;0;6;-9;0;</m:tr>
    <m:tr>0.009;115;-100;-71;-102;-10;-1;11;1;
      -6;-8;5;6;-5;-9;2;3;-2;-5;6;6;-5;-9;</m:tr>
  </m:Process>
</soap:Body>
...

```

Listing 2. An example of the body of a SOAP message for a multi-stroke character

```

...
<soap:Body xmlns:m="http://www.inkml.org/processing">
  <m:Response>
    <m:Unicode>0030, 004F, 006F</m:Unicode>
  </m:Response>
</soap:Body>
...

```

Listing 3. An example of the body of a SOAP response from the recognition service

When the recognition is incorrect, the user can fix the result on the canvas. A correction message is sent from the canvas to the recognizer and the database, see Figure 11. The correction message may contain Unicode value of the new character and the ID of the sample. After correction, if the recognition engine is context-sensitive, neighboring characters can be reclassified. Implementation of sensitivity to the context depends on the domain. With handwritten text, this task is solved by comparing a recognized word with entries in a dictionary. With mathematics, it is a harder problem, since expressions are represented as trees. Progress can be achieved by considering the most popular expressions in the subject and their empirical or grammatical properties, see for example 8.

3.2 Manipulation of Clouds

With the discussed representation of samples as clouds in high dimensional space, they can also be treated as sets. In this context, corresponding theoretical domains become applicable, such as the set theory or some elements of computational geometry. Consider training characters from two classes, say i and j , forming sets S_i and S_j respectively. Then $S_i \cap S_j$ will produce samples written in an ambiguous way: If classes i and j represent characters 9 and q then a sample that belongs to both classes can look as the one shown in Figure 3. A naïve approach to compute such intersection is to find the subset of points in each cluster with the distance to the second cluster being zero. To make the clouds linearly separable, the samples that belong to both clusters can be deleted or



Fig. 3. A sample that belongs to classes “q” and “9”

assigned a specific label. A similar operation is to find $S_q \setminus S_9$ that will result in points that can not be confused with the adjacent class.

Another example is computing the “average” character, as the center of mass of samples in a style, and using the character in calligraphic rendering of recognized samples.

These and other operations can be expressed naturally as operations on the classes represented as clouds of points. With some other machine learning frameworks the analogous procedures can be more awkward.

4 Implementation

From a high level viewpoint, the system contains the following parts, as shown in Figure 4

- A user interface for training (used to collect profiles of characters).
- A user interface for recognition (ink canvas, HLR, and recognizer).
- A cloud – a web infrastructure that serves as a recognizer (in the remote recognition mode) and as an efficient storage of user-specific training data, allowing access, update, sharing, continuous adaptation of the shapes of clusters, etc. In the current prototype implementation, the back end consists of a web server, an application server, and a DBMS.

Communication between the client application for training and the cloud is performed through sending profiles, i.e. zipped XML documents that contain personal catalogs (clouds of points). The application server communicates with the database through SQL.

4.1 Initial Training

In an adaptive recognition environment, the training phase is not required. However, having some number of training samples in each class can significantly improve the initial recognition. Training is normally performed before usage of the application or after introducing a new character to the repository. Once training is finished the profile is synchronized with the cloud.

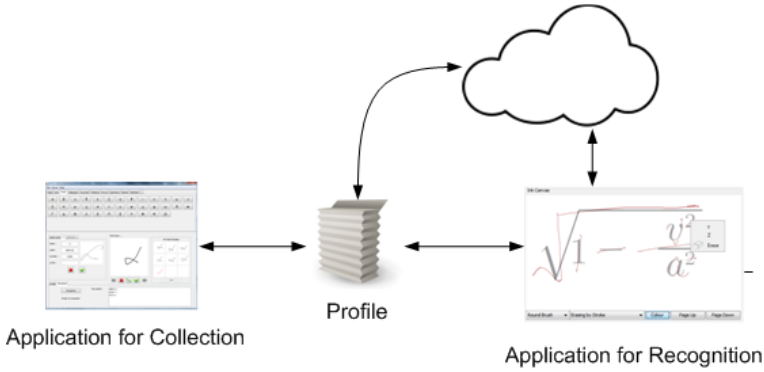


Fig. 4. Interaction of user interfaces for collection and recognition with the cloud

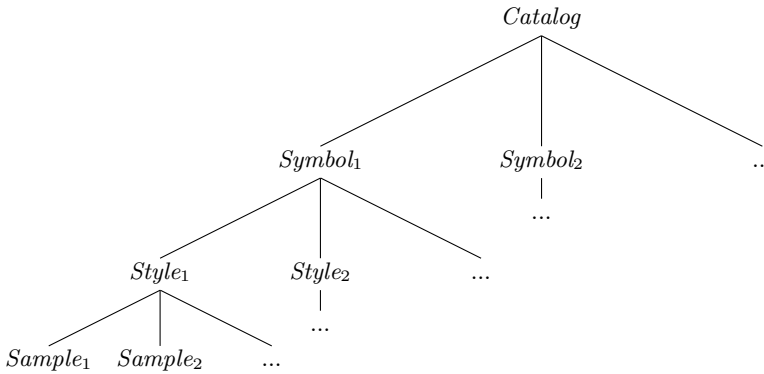


Fig. 5. The structure of a catalog

The Structure of a User Profile. A profile is a dataset of training characters used in recognition. The dataset is a collection of catalogs. Each catalog is a hierarchical container of symbols, styles, and samples. Figure 5 shows a structure of a catalog where

- *Catalog* is a catalog of related symbols, e.g. Latin characters, digits, mathematical operators, etc.
- *Symbol_i* is a recognition class, e.g. “a”, “1” or “±”.
- *Style_i* is a style, i.e. one of the possible ways to write the symbol. Our recognition algorithm is dependent on the direction of writing and the number of pen-ups of a character. For example, symbol *l* can have two styles: one style represents writing the character from the top to the bottom and another style – from the bottom to the top.
- *Sample_i* is a training sample, written according to the corresponding style.

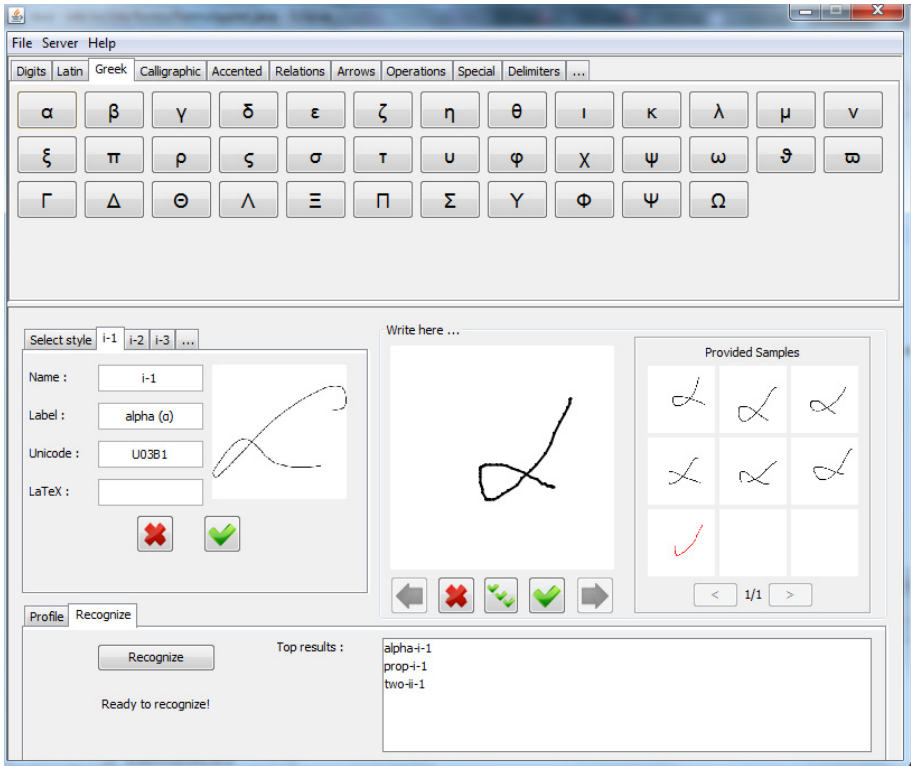


Fig. 6. The main window of the training application

Each user can have several profiles used together or independently, representing, for example, different areas of mathematics, chemistry or music. *System* profiles should also be available – the default collections of typical symbols, styles, and samples in a domain.

The XML tree of a profile corresponds to the hierarchy of a catalog: It should contain symbols, styles, samples, and coefficients. The normalized coefficients $c_i \in [-1, 1]$ can be compactly stored in a byte variable as $[127c_i]$, where $[x]$ is rounding of x to an integer [4].

4.2 Implementation of the Application

For simplicity, our current model is implemented in three-tier architecture. The client applications for collection, recognition, and the application server have been developed in Java. Requests to the application server are routed through a web server.

Client Application for Collection of Characters. The front end provides a convenient interface for the user to input and manage training samples. The

interface comes along with the structure of the user profile. Specifically, the main window of the application is a tabbed panel with each tab representing a catalog of samples, as shown in Figure 6. A tab contains a list of symbols of the catalog. Once the user selects a symbol, the panel with styles becomes available. Styles are shown as animated images for visualization of stroke order and direction. The discussed elements of the interface (catalogs, symbols, styles, and samples) are highly dynamic: A context menu is available that allows to create, to delete or to merge with another element. A profile can be saved on a local hard drive and reopened, as well as synchronized with the server.

Each provided sample should be assigned to a style. If a style has not been selected, it is determined automatically based on its shape and the number of strokes. This recognition is usually of high accuracy, since the candidate classes are styles of the selected symbol and the number of styles is typically small.

The Client Interface for Recognition. Classification of handwritten characters takes place when a user performs handwritten input through a separate application. The current implementation is integrated with the InkChat [5], a whiteboard software that facilitates engineering, scientific, or educational pen-based collaboration online. Nevertheless, a number of alternative applications can be used as the recognition front end, e.g. MathBrush [6], a pen-based system for interactive mathematics, or MathInk [13], a mathematical pen-based plug-in that can run inside computer algebra systems, such as Maple [9], or document processing software, such as Microsoft Word.

There can be two approaches to recognition – character-at-a-time (each character is recognized as it is written) and formula-at-a-time (characters are recognized in a sequence, taking advantage of the context and common deformation of samples). Classification results can be displayed super-imposed on the digital ink or replace it. For each entered character, a context menu is available that lists the top recognition candidates, as shown in Figure 7. If the user chooses another class from the candidates listed in the context menu, adjacent characters should be reclassified based on the new context information.

The Server Side. The server side has the following interacting parts: the Apache web server, an application server, and MySQL DBMS. The user uploads a profile to the application server as a zipped file. The profile is unzipped and parsed. Information is inserted in the database.

Upon download of a profile, the process is reversed – the user sends a request to the application server over the web server. The application server selects data from the database, forms an XML profile, performs compression, and sends it to the client.

In the current implementation, a client communicates with the application server over HTTP, but an encrypted communication channel is suggested in a production environment. Furthermore, profiles are recommended to be stored in the database in an encrypted format.

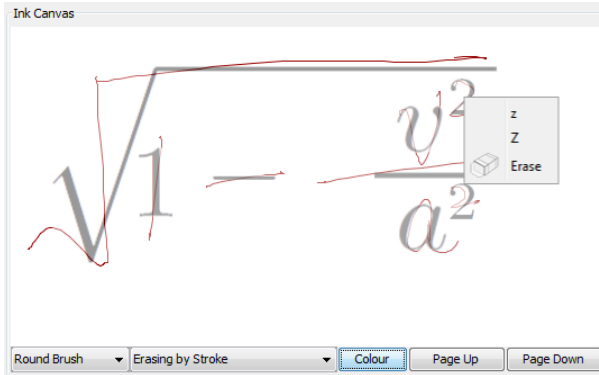


Fig. 7. Client interface for recognition

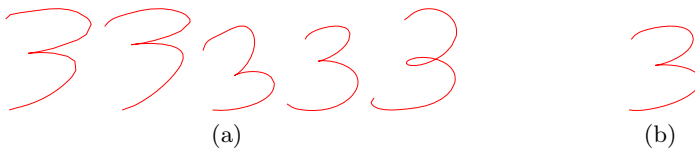


Fig. 8. (a) A set of provided samples, and (b) the average sample

4.3 Attractive Display of Recognized Characters

Some research has shown that averaging can be used to make faces look attractive [12]. We adopt a similar approach to generate visually appealing output. The shape of each output stroke is obtained by taking the average of coefficients of approximation of corresponding strokes of samples in the style

$$\bar{c}_i = \frac{\sum_{j=1}^n c_{ij}}{n}$$

where \bar{c}_i is the i -th average coefficient of a stroke and n is the number of samples in the style. The traces of the average character are then computed from the average series. This approach allows personalized output, representing samples in a visually appealing form and yet preserving the original style of the writer, as illustrated in Figure 8.

5 Experimental Evaluation

We describe results of an experiment that shows performance of adaptive author-centered recognition that can be implemented with the cloud infrastructure. The experimental setting aims to simulate decrease in the classification error depending on a user's input size, given that the application is initially trained with a default dataset.

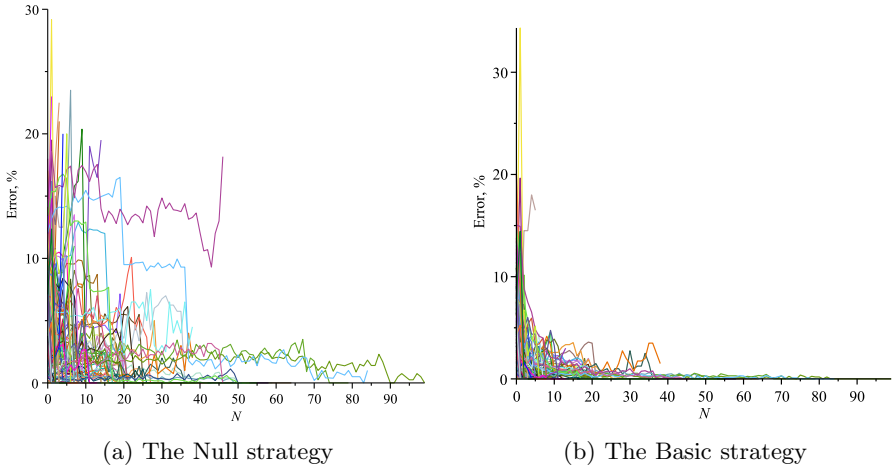


Fig. 9. The average recognition error of the $(N+1)$ -th sample in a class among all classes by an author. All authors are shown in the plot.

5.1 Setting

The experimental dataset is identical to the one described in [4]. Further, each sample is assigned to one of the 369 authors. Then for each author, the dataset is split in two parts: samples provided by the author (used in testing) and the rest of the dataset (used in training). A test sample is extracted from a randomly chosen class among those written by the test author and recognized. The recognition error of the N -th sample by the author is computed as the ratio of the number of misrecognitions of the N -th sample to the total number of N -th samples tested. This run is repeated 200 times and the average is reported. We consider two strategies for processing the recognized character

- *Null* strategy: The test sample is disregarded after recognition. This strategy is implemented for comparison with the Basic strategy.
- *Basic* strategy: The test sample is added to the corresponding training class. This facilitates adaptive recognition when the training cluster is adjusted to the style of the current user with each new sample provided.

The Basic strategy does not provide a mechanism to remove training samples that have negative impact on recognition. In [11], we developed an adaptive instance-based classifier that assigns a dynamic weight to each training exemplar. If the exemplar participates in a correct (incorrect) classification, the weight is increased (decreased). Samples with the minimal average weight are removed from the dataset.

5.2 Results

Figures 9(a) and 9(b) demonstrate the average recognition error of the N -th sample in a class among all classes by an author for the Null and the Basic

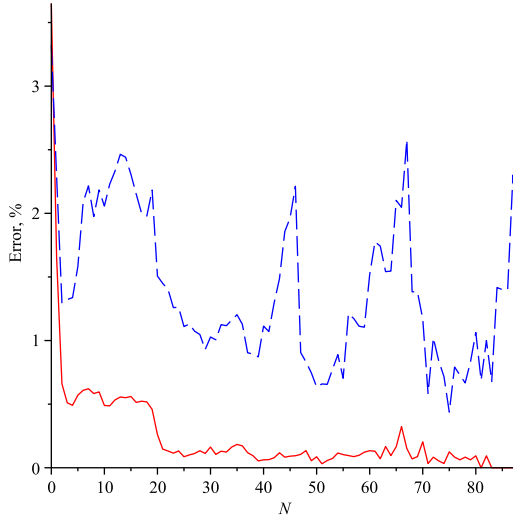


Fig. 10. The average recognition error among all authors of the $(N+1)$ -th sample in a class for the Basic strategy (solid) and the Null strategy (dash)

strategies respectively. Authors are shown in the plot in different colors. These figures show that the approach gives consistent results for different authors. The average recognition error among all authors is presented in Figure 10 for the Basic and the Null strategies.

On average, the Basic strategy demonstrates improvement over the course of use, which is most noticeable for less than 20 samples in a class by an author. Given that the dataset contains several hundred classes, synchronization of samples across devices is a valuable advantage and can make the recognition workflow efficient and smooth.

6 Conclusion

We have shown how online handwriting recognition systems can take advantage of centralized, cloud-based repositories. Incremental training data, ground truth annotations, and the machine learning framework can usefully reside on a server for the benefit of multiple client devices. We find this particularly effective for symbol sets that occur in mathematical handwriting.

With another meaning of the word “cloud”, our character recognition methods rely on clouds of points in an orthogonal series coefficient space. The representation of these clouds of training and recognition support data is quite compact, allowing collections of data sets to be cached locally even on small devices or transmitted over slow network connections. These clouds can evolve as new data is received by the server, improving recognition. These clouds also provide

a simple but effective method for handwriting neatening, by taking an average point for each style.

We find that placing recognition point sets (“clouds” in one sense) in distributed storage and computing environments (“clouds” in another sense) to be a particularly fruitful combination.

References

1. Anthony, L., Yang, J., Koedinger, K.R.: Evaluation of multimodal input for entering mathematical equations on the computer. In: CHI 2005 Extended Abstracts on Human Factors in Computing Systems, CHI EA 2005, pp. 1184–1187. ACM, New York (2005), <http://doi.acm.org/10.1145/1056808.1056872>
2. Beran, B., van Ingen, C., Fatland, D.R.: Sciscope: a participatory geoscientific web application. *Concurrency and Computation: Practice and Experience* 22(17), 2300–2312 (2010)
3. Chan, K.F., Yeung, D.Y.: Mathematical expression recognition: a survey. *IJDAR* 3(1), 3–15 (2000)
4. Golubitsky, O., Watt, S.M.: Distance-based classification of handwritten symbols. *International J. Document Analysis and Recognition* 13(2), 133–146 (2010)
5. Hu, R.: Portable implementation of digital ink: collaboration and calligraphy. Master’s thesis, University of Western Ontario, London, Canada (2009)
6. Labahn, G., Maclean, S., Marzouk, M., Rutherford, I., Tausky, D.: A preliminary report on the MathBrush pen-math system. In: Maple 2006 Conference, pp. 162–178 (2006)
7. Lamiroy, B., Lopresti, D., Korth, H., Heflin, J.: How carefully designed open resource sharing can help and expand document analysis research. In: Document Recognition and Retrieval XVIII - DRR 2011, vol. 7874. SPIE, San Francisco (2011)
8. MacLean, S., Labahn, G., Lank, E., Marzouk, M., Tausky, D.: Grammar-based techniques for creating ground-truthed sketch corpora. *Int. J. Doc. Anal. Recognit.* 14, 65–74 (2011), <http://dx.doi.org/10.1007/s10032-010-0118-4>
9. Maplesoft: Maple 13 user manual (2009)
10. Mazalov, V., Watt, S.M.: Digital ink compression via functional approximation. In: Proc. of International Conference on Frontiers in Handwriting Recognition, pp. 688–694 (2010)
11. Mazalov, V., Watt, S.M.: A structure for adaptive handwriting recognition. In: Proc. of the International Conference on Frontiers in Handwriting Recognition (submitted, 2012)
12. Perrett, D., May, K., Yoshikawa, S.: Facial shape and judgments of female attractiveness. *Nature* 368, 239–242 (1994)
13. Smirnova, E., Watt, S.M.: Communicating mathematics via pen-based computer interfaces. In: Proc. 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2008), pp. 9–18. IEEE Computer Society (September 2008)
14. Szalay, A.S.: The sloan digital sky survey and beyond. *SIGMOD Rec.* 37, 61–66 (2008), <http://doi.acm.org/10.1145/1379387.1379407>

A Web Interface for Matita^{*}

Andrea Asperti and Wilmer Ricciotti

Department of Computer Science, University of Bologna
{[aspersi](mailto:aspersi@cs.unibo.it),[ricciott](mailto:ricciott@cs.unibo.it)}@cs.unibo.it

This article describes a prototype implementation of a web interface for the Matita proof assistant [2]. The motivations behind our work are similar to those of several recent, related efforts [7,9,11,8] (see also [6]). In particular:

1. creation of a web collaborative working environment for interactive theorem proving, aimed at fostering knowledge-intensive cooperation, content creation and management;
2. exploitation of the markup in order to enrich the document with several kinds of annotations or active elements; annotations may have both a presentational/hypertextual nature, aimed to improve the quality of the proof script as a human readable document, or a more semantic nature, aimed to help the system in its processing (or re-processing) of the script;
3. platform independence with respect to operating systems, and wider accessibility also for users using devices with limited resources;
4. overcoming the installation issues typical of interactive provers, also in view of attracting a wider audience, especially in the mathematical community.

The second part of point 2. above is maybe the most distinctive feature of our approach, and in particular the main novelty with respect to [7].

In fact, delivering a proof assistant as a web application enables us to exploit the presentational capabilities of a web browser with little effort. Purely presentational markup does not require any special treatment on the part of the prover and is natively supported by the web browser. However, having an easy access to HTML-like markup allows much more flexibility. Not only can we decorate comments by means of textual formatting or pictures; executable parts of scripts reference concepts defined elsewhere, either in the same script or in the library, using possibly overloaded identifiers or notations: it is natural to enrich those identifiers with hyperlinks to the associated notions. This association is actually computed by the system every time the script is parsed, hence it is the system's job to enrich the script accordingly. Since computing associations of identifiers to library notions can be expensive, it is natural to have the system use such hyperlinks to speed up the execution of the script. Moreover, when the source text is particularly ambiguous, hyperlinks provide essential semantic information to avoid asking the user for explicit disambiguation every time the script is executed.

^{*} The project CerCo acknowledges the financial support of the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under FET-Open grant number: 243881.

Hyperlinks are an example of a textual annotation having both a presentational and a semantic value. The text enriched with hyperlinks not only provides a more dynamic and flexible format to access the library, but is also a more explicit and hence more robust representation of the information.

A further use of markup is to attach to the script information that is valuable to the system, but is not thought to be normally read by the user. This is technically a kind of presentational markup, used to hide parts of the script rather than for decorating text.

Our current implementation supports three categories of markup:

- standard HTML markup, used to add formatting to text; formatted text is currently assumed to occur in Matita comments;
- hyperlinks to Matita definitions, typically produced by the system and reused on a new parsing of the script to avoid a second disambiguation of the input (at the time of the submission, traversing hyperlinks is not yet supported, but implementing it does not look problematic);
- markup wrapping traces of execution of automation steps in the script, produced by the system on a first execution and granting a notable speed-up on future executions; the trace is normally transparent to the user, but visible on demand.

Structure of the System

Matita core The server runs a minimally reworked version of the Matita engine, equivalent to its stand-alone counterpart, but for the following features:

- the status of Matita includes the user id of its owner, as needed by an inherently multi-user web application: this allows the system to run at the same time several user-specific versions of the library;
- the lexical analyzer and the parser take into account the script markup;
- the disambiguation engine and the automation tactic produce and return information suitable for enriching the script.

For what concerns the lexical analyzer, producing specific tokens for the markup would require major modifications to the parser, which in Matita is a complex component extensible at runtime with user provided notations. In an effort to keep the parser as untouched as possible, the token stream returned by our lexical analyzer ignores the markup; however, hyperlinks that can be used for immediate disambiguation are stored in an additional table that is later accessible to the parser, which is then able to build a disambiguated abstract syntax tree (AST) for it. In order for this technique to work, we assume that disambiguation markup is only located around “leaves” of the AST (and in particular, identifiers or symbols); at the moment, this assumption does not seem to be restrictive.

Markup for automation traces, which is used only to hide additional arguments to the automation tactic, is completely handled by the user interface and can thus be safely ignored by the lexical analyzer and the parser.

Matita web daemon. The Matita web daemon is a specialized HTTP server, developed using the `Netplex` module of the Ocamlnet library¹, providing remote access to the Matita system. It exports several services:

- storage of user accounts and authentication;
- storage of user libraries;
- synchronization of user libraries with the shared library via `svn`;
- remote execution of scripts.

Such services are invoked through a CGI interface and return XML documents encoding their output.

Remote execution of scripts allows a user authoring a script on a web browser to send it to the server for processing. The typical interactions with a script are allowed, in the style of Proof-General⁴ and similarly to⁷: executing one step (tactic or directive) or the whole script, as well as undoing one step or the whole script (execution of a script until a given point is reached is performed by the client by multiple calls to single-step execution).

Parsing of the script is performed on the server, as client-side parsing of the extensible syntax used by Matita is essentially unfeasible. To execute (part of) a script, the server needs thus to receive all of the remaining text to be parsed, because the end of the next statement is not predictable without a full parsing. The Matita daemon will answer such a request by returning to the client

- the length of the portion of the original script that has been successfully executed;
- a (possibly empty) list of parsed statements, which have been enriched with mechanically generated markup including disambiguation hints and automation traces (the length of this updated text does *not* match the previous value in general);
- an HTML representation of the proof state of the system after the execution of the last statement (if the execution stopped in the middle of a proof);
- a representation of the error that prevented a further execution of the script (if the execution stopped because of an error).

Collaborative formalization. The daemon provides a preliminary support for collaborative formalization, currently coming in the form of a centralized library maintained by means of `svn`. Other authors (see¹¹) have advocated the use of distributed versioning systems (e.g. `Git`). Our choice is mainly related to the reuse of the original Matita repository and to the fact that `svn` already supports the kind of distributed activity we have in mind. The effective usability and scalability of this approach will be tested in the future.

The client. The Matita web client (Figure¹¹) was initially written in plain Javascript and is currently being ported to the jQuery² framework. The client

¹ <http://projects.camlcity.org/projects/ocamlnet.html>

² <http://jquery.com>

implements a user interface that is essentially similar to the one of ProofGeneral [4], CtCoq and CoqIDE [5], or stand-alone Matita [3], but in the form of a web page. This includes displaying the script (disabling editing for the already executed part), buttons for script navigation, boxes for proof state (including multiple open goals) and disambiguation, instant conversion of $\text{T}_{\text{E}}\text{X}$ -like escapes to Unicode symbols, and essential interface for accessing the remote file system.

The implementation issues are similar to those described in [10]. The web interface does not need to understand much of Matita: information like being in an unfinished proof or in disambiguation mode can be easily inferred from the data structures returned from the server. On the other hand, some code is necessary to convert Matita markup to HTML markup and vice-versa.

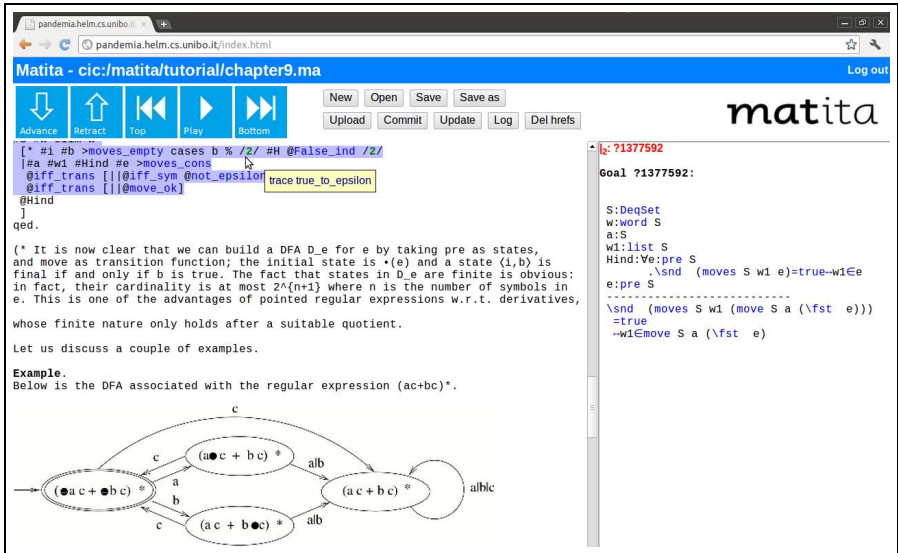


Fig. 1. MatitaWeb in action

Availability. The Matita web interface is accessible from the website <http://pandemia.helm.cs.unibo.it/login.html>. Accounts for accessing the interface are provided by the authors on request.

References

1. Alama, J., Brink, K., Mamane, L., Urban, J.: Large Formal Wikis: Issues and Solutions. In: Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.) MKM 2011 and Calculemus 2011. LNCS, vol. 6824, pp. 133–148. Springer, Heidelberg (2011)
2. Asperti, A., Ricciotti, W., Sacerdoti Coen, C., Tassi, E.: The Matita Interactive Theorem Prover. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 64–69. Springer, Heidelberg (2011)

3. Asperti, A., Sacerdoti Coen, C., Tassi, E., Zacchiroli, S.: User interaction with the Matita proof assistant. *Journal of Automated Reasoning* 39(2), 109–139 (2007)
4. Aspinall, D.: Proof General: A Generic Tool for Proof Development. In: Graf, S. (ed.) TACAS 2000. LNCS, vol. 1785, pp. 38–43. Springer, Heidelberg (2000)
5. Bertot, Y., Théry, L.: A generic approach to building user interfaces for theorem provers. *Journal of Symbolic Computation* 25, 161–194 (1998)
6. Geuvers, H.: Proof Assistants: history, ideas and future. *Sadhana* 34(1), 3–25 (2009)
7. Kaliszyk, C.: Web interfaces for proof assistants. *Electr. Notes Theor. Comput. Sci.* 174(2), 49–61 (2007)
8. Tankink, C., Geuvers, H., McKinna, J., Wiedijk, F.: Proviola: A Tool for Proof Re-animation. In: Autexier, S., Calmet, J., Delahaye, D., Ion, P.D.F., Rideau, L., Rioboo, R., Sexton, A.P. (eds.) AISC 2010. LNCS, vol. 6167, pp. 440–454. Springer, Heidelberg (2010)
9. Urban, J., Alama, J., Rudnicki, P., Geuvers, H.: A wiki for mizar: Motivation, considerations, and initial prototype. *CoRR*, abs/1005.4552 (2010)
10. Wenzel, M.: Isabelle as Document-Oriented Proof Assistant. In: Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.) MKM 2011 and Calculemus 2011. LNCS, vol. 6824, pp. 244–259. Springer, Heidelberg (2011)

MaxTract: Converting PDF to \LaTeX , MathML and Text

Josef B. Baker, Alan P. Sexton, and Volker Sorge

School of Computer Science, University of Birmingham

`{j.baker,a.p.sexton,v.sorge}@cs.bham.ac.uk`

<http://www.cs.bham.ac.uk/~{jbb|aps|vxs}>

1 Introduction

In this paper we present the first public, online demonstration of MaxTract; a tool that converts PDF files containing mathematics into multiple formats including \LaTeX , HTML with embedded MathML, and plain text. Using a bespoke PDF parser and image analyser, we directly extract character and font information to use as input for a linear grammar which, in conjunction with specialised drivers, can accurately recognise and reproduce both the two dimensional relationships between symbols in mathematical formulae and the one dimensional relationships present in standard text.

The main goals of MaxTract are to provide translation services into standard mathematical markup languages and to add accessibility to mathematical documents on multiple levels. This includes both accessibility in the narrow sense of providing access to content for print impaired users, such as those with visual impairments, dyslexia or dyspraxia, as well as more generally to enable any user access to the mathematical content at more re-usable levels than merely visual. MaxTract produces output compatible with web browsers, screen readers, and tools such as copy and paste, which is achieved by enriching the regular text with mathematical markup. The output can also be used directly, within the limits of the presentation MathML produced, as machine readable mathematical input to software systems such as Mathematica or Maple.

2 MaxTract Process

Although the PDF documents that MaxTract works on are electronic documents with character and font information, actually extracting that information and then analysing it to construct an interpretation of the text and mathematical formulae contained is a somewhat involved process.

We start by using image analysis over an input file rendered to TIF to identify the precise bounding boxes of the glyphs, or connected components, on a page. This is necessary as the PDF format does not encode this information but precise bounding box information for the characters is critical to the two dimensional analysis necessary for mathematical formula recognition. The bounding boxes are mapped to the character and font information extracted via PDF analysis

to produce a list of symbols with their detailed location and bounding box information. The extraction is completed by parsing the content streams and font objects comprising a PDF file with a bespoke PDF parser we have written, based upon the PDF specification [1]. A symbol consists of a name, bounding box, base point, font size and font name. Projection profile cutting is then used to identify lines of symbols which are passed to a linear grammar as lists of symbols.

The linear grammar creates a parse tree based upon the two dimensional relationships of the symbols and their appearance. The grammar has been designed to produce a parse tree rich enough to be translated by specialised drivers to produce a wide variety of output in various markup. The extraction process, analysis and grammar are explained in detail in [3,4].

3 Translation

We use three main output drivers to produce markup, namely a L^AT_EX, MathML and plain text driver. We combine these drivers in a number of ways to produce various output formats designed to be accessible to users with a wide variety of software. Here we explain the basic drivers.

3.1 Basic Drivers

Each of the basic drivers are used in conjunction with a layout analysis module, which identifies structures such as display mathematics, alignment and justification, columns and paragraphs.

L^AT_EX. This produces a `.tex` file which, when compiled, has been designed to reproduce the original formatting and style closely. All of the fonts identified from the original PDF file are reused, and reproduced together with layout and spacing where possible.

MathML. The MathML driver returns an `.xhtml` file, containing standard HTML, interspersed with presentation MathML where appropriate. Unlike the L^AT_EX driver, styling is not closely reproduced, with standard HTML fonts and formatting used instead of the originals, however elements such as headings and paragraphs are retained.

Festival. The festival driver produces plain text that can be given to text-to-speech engines directly. We have particularly focused on and experimented with Festival [5], and have added some optimisation for this particular engine.

3.2 Annotated PDF

The idea of annotated PDFs is not only to reproduce the original document using the L^AT_EX driver, but also allow the user to view and copy the markup for each mathematical formula when viewing the compiled PDF. By associating each

formula with a `\pdfannot` command, which is processed by `pdfflatex`, the user is presented with clickable notes containing the respective markup. These notes can be opened, closed, moved and edited by the user, and of course their content can be viewed and copied. An example is shown in Figure 1. The annotation features are not universally supported by PDF browsers, although they are part of the PDF specification and supported by Adobe Reader [2].

$$e^{-n} \sum_{k=0}^{\infty} \frac{n^k}{k!} |k - n| \leq \sqrt{n}.$$


The image shows a screenshot of a PDF annotation window. The window title is "LaTeX Code" and it has a date and time stamp "18/04/12 17:07:49". There are "Options" and "Close" buttons. The main content of the window is the LaTeX code for the equation shown to the left: `e^{[-n]} sum^{[infinity]}_{[k=0]} \frac{n^{[k]} }{[k!]} |k - n| \leq \sqrt{[n]}.`

Fig. 1. Equation in a PDF file annotated with \LaTeX

LaTeX Annotations. A special version of the \LaTeX driver is used to produce the \LaTeX annotations. As the markup is designed to be viewed, copied and possibly edited, positioning and font commands are removed in order to improve the clarity and simplicity of the generated code.

MathML Annotations. These are produced by the same MathML driver as described previously, containing valid snippets of MathML for each formula.

Double Annotations. In addition to the singly annotated files we can also produce doubly annotated files, where each formula is associated with both the MathML and \LaTeX .

3.3 Layered PDF

By taking advantage of the Optional Content features of PDF, we can create files that contain multiple layers, allowing a user to switch between, say the rendered file and the underlying \LaTeX . To achieve this we make use of two additional packages, `OCG` [7] and `textpos` [6], the first of which is used to produce the separate layers and the second to position each layer correctly. Again these official features of PDF are not widely supported by PDF browsers other than Adobe Reader. Figure 2 shows the layer choice dialog in Adobe Reader with the various layers available in a document produced by MaxTract.

Text Layer. This layer is produced by using the festival driver. The layer has been designed to work with the read-out-loud facility in Adobe Reader, allowing the screen reader to accurately read the whole document, including mathematics which is usually garbled in standard documents.

LaTeX Code Layer. In a similar manner to the annotated PDF, this allows the user to see and copy the underlying \LaTeX code. However, the layer shows the code for the whole page rather than just the formulae. This is the simple \LaTeX code, without fonts and spacing commands.

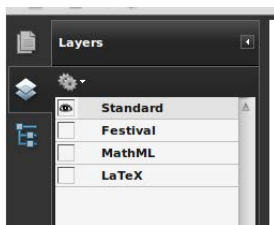


Fig. 2. Selection of layers in a PDF file

Both Layers. Again, we can also produce double layered PDF files containing both the L^AT_EX and text.

3.4 Accessibility Formats

The accessibility formats that we produce are designed to be compatible with all screen readers. This is achieved by producing standard, plain text files, with none of the special characters or formatting that are often incompatible with accessibility tools. Any non ASCII symbols, or groups of symbols with non-linear relationships are replaced with alternative ASCII based text.

Text Only. This is the direct output of the festival driver, producing a plain text file that can be given to text-to-speech engines.

Text Only as Latex. Text only as L^AT_EX wraps the text described above with a standard L^AT_EX header and footer so that it can be compiled into a PDF file. No other commands are used and any mathematics is replaced by alternative text. This is essentially the same as the text layer from the layered PDF.

Text Only as HTML. This produces a standard .html file to be used with speech enabled browsers. The text is wrapped in HTML with a standard header and footer, and line break is the only tag used. Mathematical equations are replaced by in line alternative text.

4 MaxTract Online Interface

The MaxTract demonstration consists of an HTML form to select and upload a PDF file for extraction, select an output format and enter an email address. It can be found at <http://www.cs.bham.ac.uk/research/groupings/reasoning/sdag/maxtract.php>.

A PDF file is compatible with MaxTract if it contains only fonts and encodings that are embedded and of type 1. This can be checked by viewing the fonts tab in the file properties within Adobe Reader. Once uploaded, the file is processed if it is found to be compatible with MaxTract, and the user is emailed with a link

to download the output. If the file cannot be processed, this will be confirmed via email.

An example of each type of output is also available to be viewed or downloaded from the MaxTract web site. As stated in section 3, some of these formats make use of advanced features of PDF which are not supported by all readers, however they are compatible with Adobe Reader which should be used to view our output.

References

1. Adobe. PDF Reference fifth edition Adobe Portable Document Format Version 1.6. Adobe Systems (2004)
2. Adobe. Adobe Reader X. Adobe Systems (2012), <http://get.adobe.com/uk/reader/>
3. Baker, J.B., Sexton, A.P., Sorge, V.: A Linear Grammar Approach to Mathematical Formula Recognition from PDF. In: Carette, J., Dixon, L., Coen, C.S., Watt, S.M. (eds.) MKM 2009, Held as Part of CICM 2009. LNCS (LNAI), vol. 5625, pp. 201–216. Springer, Heidelberg (2009)
4. Baker, J.B., Sexton, A.P., Sorge, V.: Towards reverse engineering of PDF documents. In: Sojka, P., Bouche, T. (eds.) Towards a Digital Mathematics Library, DML 2011, Bertinoro, Italy, pp. 65–75. Masaryk University Press (July 2011)
5. Black, A.W., Taylor, P.A.: The Festival Speech Synthesis System: System documentation. Technical Report HCRC/TR-83, Human Communication Research Centre, University of Edinburgh, Scotland, UK (1997), <http://www.cstr.ed.ac.uk/projects/festival.html>
6. Gray, N.: Textpos (2010), <http://purl.org/nxg/dist/textpos>
7. Marik, R.: OCGtools (2012), <http://ctan.org/pkg/ocgtools>

New Developments in Parsing Mizar*

Czesław Bylinski and Jesse Alama

Center for Artificial Intelligence
New University of Lisbon
Portugal
j.alama@fct.unl.pt

Abstract. The Mizar language aims to capture mathematical vernacular by providing a rich language for mathematics. From the perspective of a user, the richness of the language is welcome because it makes writing texts more “natural”. But for the developer, the richness leads to syntactic complexity, such as dealing with overloading.

Recently the Mizar team has been making a fresh approach to the problem of parsing the Mizar language. One aim is to make the language accessible to users and other developers. In this paper we describe these new parsing efforts and some applications thereof, such as large-scale text refactorings, pretty-printing, HTTP parsing services, and normalizations of Mizar texts.

1 Introduction

The Mizar system provides a language for declaratively expressing mathematical content and writing mathematical proofs. One of the principal aims of the Mizar project is to capture “the mathematical vernacular” by permitting authors to use linguistic constructions that mimic ordinary informal mathematical writing. The richness is welcome for authors of Mizar texts. However, a rich, flexible, expressive language is good for authors can lead to difficulties for developers and enthusiasts. Certain experiments with the Mizar language and its vast library of formalized mathematical knowledge (the Mizar Mathematical Library, or MML), naturally lead to rewriting Mizar texts in various ways. For some purposes one can work entirely on the semantic level of Mizar texts; one may not need to know precisely what the source text is, but only its semantic form. For such purposes, an XML presentation of Mizar texts has long been available [6]. However, for some tasks the purely semantic form of a Mizar text is not what is wanted. Until

* Supported by the ESF research project *Dialogical Foundations of Semantics* within the ESF Eurocores program *LogICCC* (funded by the Portuguese Science Foundation, FCT LogICCC/0001/2007). Research for this paper was partially done while a visiting fellow at the Isaac Newton Institute for the Mathematical Sciences in the program ‘Semantics & Syntax’. Karol Pałk deserves thanks for his patient assistance in developing customized Mizar text rewriting tools.

recently there has been no standalone tool, distributed with Mizar, that would simply parse Mizar texts and present the parse trees in a workable form.¹

Parsing texts for many proof assistants is often facilitated through the environment in which these proof assistants are executed. Thus, texts written for those systems working on top of a Lisp, such as IMPS, PVS, and ACL2, already come parsed, so one has more or less immediate access to the desired parse trees for terms, formulas, proofs, etc. Other systems, such as Coq and HOL light, use syntax extensions (e.g., Camlp4 for Objective Caml) to “raise” the ambient programming language to the desired level of proof texts. For Mizar, there is no such ambient environment or read-eval-print loop; working with Mizar is more akin to writing a C program or \LaTeX document, submitting it to *gcc* or *pdflatex*, and inspecting the results.

This paper describes new efforts by the Mizar team to make their language more workable and illustrates some of the fruits these efforts have already borne. This paper does not explain *how* to parse arbitrary Mizar texts. And for lack of space we cannot go into the detail about the Mizar system; see [3,4].

In Section 2, we discuss different views of Mizar texts that are now available. Section 3 describes some current applications made possible by opening up Mizar texts, and describes some HTTP-based services for those who wish to connect their own tools to Mizar services. Section 4 concludes by sketching further work and potential applications.

2 Layers of a Mizar Text

It is common in parsing theory to distinguish various analyses or layers of a text, considered in the first place as a sequence of bytes or characters [1]. Traditionally the first task in parsing is **lexical analysis** or **scanning**: to compute, from a stream of characters, a stream of *tokens*, i.e., terminals of a production grammar G . From a stream of tokens one then carries out a **syntactic analysis**, which is the synthesis of tokens into groups that match the production rules of G .

One cannot, in general, lexically analyze Mizar texts without access to the MML. Overloading (using the same symbol for multiple, possibly unrelated meanings) already implies that parsing will be non-trivial, and overloading is used extensively in the Mizar library. Even with a lexical analysis of a Mizar text, how should it be understood syntactically? Through Mizar’s support for **dependent types**, the overloading problem is further complicated. Consider, for example, the Mizar fragment

```
let X be set,
    R be Relation of X, Y;
```

The notion of a (binary) relation is indicated by the non-dependent (zero-argument) type `Relation`. There is also the binary notion *relation whose domain is a subset of X and whose range is a subset of Y* , which is expressed

¹ One parser tool, *lisppars*, is distributed with Mizar. *lisppars* is mainly used to facilitate authoring Mizar texts with Emacs [5]; it carries out fast lexical analysis only and does not output parse trees.

as **Relation of X,Y**. Finally, we have the one-argument notion *relation whose domain is a subset of X and whose range is a subset of X* which is written **Relation of X**. In the text fragment above, we have to determine which possibility is correct, but this information would not be contained in a token stream (is Y the second argument of an instance of the binary **Relation** type, or is it the third variable introduced by the **let**?).

2.1 Normalizations of Mizar Texts

One goal of opening up the Mizar parser is to help those interested in working with Mizar texts to not have to rely on the Mizar codebase to do their own experiments with Mizar texts. We now describe two normalizations of (arbitrary) Mizar texts, which we call weakly strict and more strict. The results of these two normalizations on a Mizar text can be easily parsed by a standard LR parser, such as those generated by the standard tool *bison*² and have further desirable syntactic and semantic properties. Other normalizations beyond these two are certainly possible. For example, whitespace, labels for definitions, theorems, lemmas, etc., are rewritten by the normalizations we discuss; one can imagine applications where such information ought not be tampered with.

2.2 Weakly Strict Mizar

The aim of the weakly strict Mizar (WSM) transformation is to define a class of Mizar texts for which one could easily write an standard, standalone parser that does not require any further use of the Mizar tools. In a weakly strict Mizar text all notations are disambiguated and fully parenthesized, and all statements take up exactly one line. (This is a different transformation than single-line variant AUT-SL of the Automath system [2].) Consider:

```
reserve P,R for Relation of X,Y;
```

This Mizar fragment is ambiguous: it is possible that the variable Y is a third reserved variable (after the variables P and R), and it is possible that Y is an argument of the dependent type **Relation of X,Y**. The text becomes disambiguated by the weakly strict Mizar normalization to

```
reserve P , R for ( Relation of X , Y ) ;
```

and now the intended reading is syntactically evident, thanks to explicit bracketing and whitespace. (Any information that is implicitly contained by whitespace structure in the original text is destroyed.)

The result of the one-line approach of the weakly strict Mizar normalization is, in many cases, excessive parenthesization, unnecessary whitespace, and rather long lines.³ The point of the weakly strict Mizar normalization is not to produce

² <http://www.gnu.org/software/bison/>

³ The longest line in the “WSM-ified” library has length 6042. About 60% (to be precise, 694) of the articles in the WSM form of the current version of the Mizar Mathematical Library (4.181.1147) have lines of length at least 500 characters. The average line length across the whole “WSM-ified” library is 54.7.

attractive human-readable texts. Instead, the aim is to transform Mizar texts so that they have a simpler grammatical structure.

2.3 More Strict Mizar

A second normalization that we have implemented is called, for lack of a better term, more strict Mizar (MSM). The aim of the MSM normalization is to define a class of Mizar texts that are canonicalized in the following ways:

- From the name alone of an occurrence of a variable one can determine the category (reserved variable, free variable, bound variable, etc.) to which the occurrence belongs. (Such inferences are of course not valid for arbitrary Mizar texts.)
- All formulas are labeled, even those that were unlabeled in the original text.
- Some “syntactic sugar” is expanded.
- Toplevel logical linking is replaced by explicit reference. Thus,

```
 $\phi$ ; then  $\psi$ ;
```

using the keyword **then** includes the previous statement (ϕ) as the justification of ψ . Under the MSM transformation, such logical relationships are rewritten as

```
Label1:  $\phi$ ;  
Label2:  $\psi$  by Label1;
```

Now both formulas have new labels **Label1** and **Label2**. The logical link between ϕ and ψ , previously indicated by the keyword **then**, is replaced by an explicit reference to the new label (**Label1**) for ϕ .

- All labels of formulas and names of variables in a Mizar are serially ordered.

MSM Mizar texts are useful because they permit certain “semantic” inferences to be made simply by looking at the syntax. For example, since all formulas are labeled and any use of a formula must be done through its label, one can infer simply by looking at labels of formulas in a text whether a formula is used. By looking only at the name of a variable, one can determine whether it was introduced inside the current proof or was defined earlier.

3 Applications

Opening up the Mizar parser by providing new tools that produce parse trees naturally suggests further useful text transformations, such as pretty printing. An HTTP parsing service for these new developments is available for public consumption. Four services are available. Submitting a suitable **GET** request to the service and supplying a Mizar text in the message body, one can obtain as a response the XML parse tree for the text, a pretty-printed form of it, or the WSM or MSM form of a text (either as plain text or as XML). The HTTP services permit users to parse Mizar texts without having access to the MML, or even the Mizar tools. See

<http://mizar.cs.ualberta.ca/parsing/>

to learn more about the parsing service, how to prepare suitable HTTP parsing requests, and how to interpret the results.

4 Conclusion and Future Work

Parsing is an essential task for any proof assistant. In the case of Mizar, parsing is a thorny issue because of the richness of its language and its accompanying library. New tools for parsing Mizar, with an eye toward those who wish to design their own Mizar applications without (entirely) relying on the Mizar tools, are now available. Various normalizations for Mizar texts have been defined. Further useful normalizations are possible. At present we are experimenting with a so-called “without reservations” Mizar (WRM), in which there are no so-called reserved variables; in WRM texts the semantics of any formula is completely determined by the block in which it appears, which should make processing of Mizar texts even more efficient.

References

1. Aho, A., Lam, M., Sethi, R., Ullman, J.: *Compilers: Principles, Techniques, and Tools*. Pearson/Addison Wesley (2007)
2. de Bruijn, N.G.: AUT-SL, a single-line version of Automath. In: Nederpelt, R., Geuvers, J.H., de Vrijer, R.C. (eds.) *Selected Papers on Automath*. *Studies in Logic and the Foundations of Mathematics*, vol. 133, ch. B.2, pp. 275–281. North-Holland (1994)
3. Grabowski, A., Kornilowicz, A., Naumowicz, A.: Mizar in a nutshell. *Journal of Formalized Reasoning* 3(2), 153–245 (2010)
4. Matuszewski, R., Rudnicki, P.: Mizar: The first 30 years. *Mechanized Mathematics and its Applications* 4(1), 3–24 (2005)
5. Urban, J.: MizarMode—An integrated proof assistance tool for the Mizar way of formalizing mathematics. *Journal of Applied Logic* 4(4), 414–427 (2006), <http://www.sciencedirect.com/science/article/pii/S1570868305000698>
6. Urban, J.: XML-izing Mizar: Making Semantic Processing and Presentation of MML Easy. In: Kohlhase, M. (ed.) *MKM 2005*. LNCS (LNAI), vol. 3863, pp. 346–360. Springer, Heidelberg (2006)

Open Geometry Textbook: A Case Study of Knowledge Acquisition via Collective Intelligence (Project Description)

Xiaoyu Chen¹, Wei Li¹, Jie Luo¹, and Dongming Wang²

¹ SKLSDE – School of Computer Science and Engineering, Beihang University, Beijing 100191, China

² Laboratoire d’Informatique de Paris 6, Université Pierre et Marie Curie – CNRS, 4 place Jussieu – BP 169, 75252 Paris cedex 05, France

Abstract. We present an Open Geometry Textbook project whose main objective is to develop a web-based platform for acquiring geometric knowledge to form a textbook via the collective intelligence of a massive number of web users. The platform will provide a formal representation language for users to contribute formalized geometric knowledge contents and will integrate software tools for automated knowledge processing and content management. Collected geometric knowledge will be presented as a dynamic textbook, which can be freely accessed, timely updated, and soundly revised by any interested web user and whose authoritatively ideal versions will be published in print. We will design and implement effective mechanisms and tools for open textbook author identification, soundness checking, revision assessment, and version maintenance.

Keywords: Geometry, knowledge acquisition, collective intelligence, open textbook, version maintenance.

1 Motivations

Textbooks, as a standard form of presentation for structured domain knowledge, have played a significant role in education and research. Classical textbooks share some common features.

- They are static documents in which domain knowledge is machine-readable rather than machine-comprehensible.
- The contents of a textbook are constructed usually by a few domain experts with their own intelligence. There is no effective mechanism for acquiring input and feedback from widespread readers.
- Textbook revision, update, and improvement are not timely, depending on the publishers and the availability of the authors and taking long periods of time in practice.

Inspired by the rapid development of web and computing technologies, the swift growth of web users (or *netizens*) around the world, and the new approach of acquiring knowledge via collective intelligence [7], we have started developing a kind of modern textbook, called *open textbook*. It is a running software system

that appears as an electronic, dynamic textbook and is freely accessible. Netizens are allowed to revise the contents of the textbook and can review the latest revisions and assess them using implemented mechanisms. When the assessments of revisions are justified, the textbook is updated automatically in real time and its new version is then published online.¹ The open textbook does not have fixed authors, is maintained by a dynamic subcommunity of netizens, and thus may live longer than a traditional textbook.

Compared with articles in Wikipedia (<http://www.wikipedia.org/>) and modules in the open educational content repository of Connexions (<http://cnx.org/>), domain knowledge collected in the open textbook will be formalized, more focused, and better structured, so software tools can be developed or integrated to automate the process of theorem proving, diagram generation, consistency and soundness checking, textbook version maintenance, etc.

Euclidean geometry has been taken for our case study because it is a rich and typical subject of mathematics involving complex knowledge of various kinds and because the geometric knowledge base [2], the formal Geometry Description Language (GDL – <http://geo.cc4cm.org/text/GDL.html>), and the prototype of an electronic geometry textbook [1] produced from our past research on EGT [3] can be used for our implementation of the open textbook project.

2 Objectives

The main objective of the Open Geometry Textbook project is to design and implement a web-based platform for acquiring geometric knowledge to form a textbook via the collective intelligence of a massive number of web users (including geometry experts, teachers, learners, and amateurs) and to study the feasibility of our open textbook approach for automated acquisition of knowledge in mathematics, with geometry as a special case. The project will also help build up a collection of geometric knowledge from netizens' contributions and an integrated environment with sophisticated software tools for interested users to access, explore, communicate, disseminate, and publish geometric knowledge. To succeed in our objectives, we will focus our research and development on the following three aspects.

Content Management. A structural knowledge base will be created to manage textbook contents (in natural languages and/or formalized representations), their revisions, and the results of assessments, so that new versions of the textbook or parts thereof may be generated by incorporating accepted revisions

¹ By *content* we mean a structural part of the textbook. It can be a chapter, a paragraph, a theorem, or an example. By *revision* we mean either a new content obtained from an existing content of the textbook by deletion and/or modification, or a newly added content. *Version* is used for the entire textbook. A new version is produced from an existing version of the textbook by incorporating accepted revisions. Creative Commons Attribution-ShareAlike license will be adopted as the main content license for the open textbook.

in real time and in different formats (e.g., PDF and HTML) for display and printing at the reader's choice.

Revision Assessment. Factors (such as correctness, logicity, and presentation) of impact on the quality of the textbook will be studied. Models and mechanisms will be introduced to measure and rate the quality of revisions. It is expected that the coverage and quality of the textbook will tend to be improved gradually as the number of its version increases.

Knowledge Processing. A language for formalizing textbook contents will be provided and most textbook contents will be formalized as possible (by motivated users and/or the developers). Several external software packages will be integrated into the textbook for automated geometric computation, theorem proving, diagram generation, and consistency and soundness checking. Practical tools will be developed to guide and assist users to formalize, structure, annotate, and validate their contributions.

3 Methodologies

Several available methodologies will be adapted for developing software components of the textbook to manage knowledge contents and to assist authors to make sound revisions, as well as readers to assess contributions. One of them we have used is to encapsulate interrelated knowledge data into knowledge objects and structure knowledge objects into knowledge graphs to realize a knowledge base for managing multiversion geometric knowledge contents [2]. We will adopt this methodology to encapsulate revisions with their corresponding assessment results for managing constantly revised contents. For presenting contents in readable formats, we have implemented interfaces to generate XML documents by assembling the required data from the knowledge base and transform them into HTML documents for display [1]. Our choice of notations for knowledge presentation will be justified by consulting the census in [6] and other references.

The language GDL proposed in our EGT project can be used to formalize geometric definitions, configurations, propositions, etc. We have implemented an interface via which formalized geometric theorems in the textbook can be proved automatically by calling provers from GEOTHER (<http://www-polsys.lip6.fr/~wang/GEOTHER/>). An interface has also been developed to generate drawing instructions in the syntax of GeoGebra (<http://www.geogebra.org/cms/>) automatically from configurations formulated in GDL [1].

GDL can serve as the basis of a formalization language for contents of the open geometry textbook. We will extend its expressibility such that theories in the textbook can be formalized by using the same language. To make consistency and soundness checking as automatic as possible, we will enlarge our collection² of formalized geometric definitions and propositions and motivate users to contribute formalized contents. When contents of the textbook are

² Currently, it contains 137 formalized definitions and 102 formalized theorems.

revised, theories in the textbook may become inconsistent. We will adapt our general-purpose implementation (<http://R.nlsde.buaa.edu.cn/>) of R-calculus [5] to produce consistent candidate theories after content revisions. Any of the candidate theories may be selected to substitute the original theory, but in this case all the proofs that use axioms or theorems not included in the new theory need be verified. Geometric proofs will be formulated with structure so that their correctness can be (partially) verified by using proof assistants (such as Isabelle – <http://www.cl.cam.ac.uk/research/hvg/Isabelle/>). For checking the consistency of presentation structure and the completeness and redundancy of textbook contents in real time, some methods have been implemented [3].

Checking the consistency and soundness of informal contents is a challenging issue. We are going to make use of some existing tools (e.g., LanguageTool – <http://www.languagetool.org/>) for language spelling and grammar checking and to develop assistants for authors to encapsulate informal and formalized contents with same meanings into individual objects interactively in order to facilitate their interplay. This work is not hard because the meanings of formalized contents can be easily interpreted. By means of such encapsulation, validation of informal contents can be reduced to consistency and soundness checking of formalized contents. The validity of informal contents for which no corresponding formalized contents are provided would depend on the knowledge level and reliability of the authors.

The open textbook will build on the basis of an open source web content management system (such as Plone – <http://plone.org/>) which has facilities for object-oriented data management, revision control, and document publication.

4 Mechanisms

Assessments of content revisions will be performed at three levels: at the first level, revisions (on formalized contents, authoring formats, language spelling and grammar, etc.) are checked and their quality is measured by using software tools (see, e.g., [4] for some measurement models); at the second level, the quality of each revision is rated by interested readers (by means of giving scores on impact factors, as in the rating system of WikiProject Mathematics/Wikipedia 1.0). An algorithm will be designed and implemented to rank versions of the textbook, generated with different accepted revisions, according to the results of assessments. The order of textbook versions itself will change dynamically. At any time, the version ranked first is accepted as a temporarily ideal version of the textbook.

At certain times (e.g., once a year or after several major revisions), assessments are done at the third level by experts, who may be authors with high reliability and expertise determined according to their past behaviors and the quality of the contributions they made. Such experts are invited to help review, revise, and improve the textbook to produce milestone versions, accepted as authoritatively ideal versions. The experts are advised to take the results of assessments from the first two levels into account, so each authoritatively ideal version has the

highest rank at the time when it is produced. Some of the authoritatively ideal versions will be published in print and thus become static.

Interested netizens are the principal working force for the construction, maintenance, improvement, and expansion of the open geometry textbook. Open textbook authors will be kept anonymous with identification codes to the public unless they choose to reveal their real names and the privacy of all personal information will be securely protected. To find good strategies to popularize our project and to attract netizens to participate as authors and readers, we will take necessary measures, for example, cooperating with teachers in selected schools and awarding authors for distinguished contributions and readers for fairest assessments. We will approach relevant organizations and project teams to get our platform linked to their webpages and find leading publishers to publish some of the authoritatively ideal versions of the textbook.

5 Timelines

We plan to build up the infrastructure for the web-based platform within one and half years. A prototype of the platform with a preliminary version of the textbook will be accessible to a limited number of netizens from early 2013. Meanwhile, supporting tools will be released. We will spend one year to test and enhance the platform and tools by soliciting input and feedback from experts in the community of mathematical knowledge management and from netizens outside this research area. An initial yet complete version of the textbook prepared by the developers will be released and made publicly accessible via the platform in early 2014. We expect that a substantially improved and expanded milestone version of the textbook with contributions from over 500 authors will be produced by the end of 2016. A webpage for the proposed project is under construction and will be available at <http://OpenText.nlsde.buaa.edu.cn/>. The reader is welcome to contribute to the project by submitting comments, suggestions, ideas, and criticisms via the webpage.

Acknowledgments. The authors wish to thank Christoph Lange and the referees for many insightful comments which have helped bring the paper to the present form. This work has been supported by the SKLSDE Open Fund SKLSDE-2011KF-02.

References

1. Chen, X.: Electronic Geometry Textbook: A Geometric Textbook Knowledge Management System. In: Autexier, S., Calmet, J., Delahaye, D., Ion, P.D.F., Rideau, L., Rioboo, R., Sexton, A.P. (eds.) MKM 2010. LNCS (LNAI), vol. 6167, pp. 278–292. Springer, Heidelberg (2010)
2. Chen, X., Huang, Y., Wang, D.: On the Design and Implementation of a Geometric Knowledge Base. In: Sturm, T., Zengler, C. (eds.) ADG 2008. LNCS (LNAI), vol. 6301, pp. 22–41. Springer, Heidelberg (2011)

3. Chen, X., Wang, D.: Management of Geometric Knowledge in Textbooks. *Data & Knowledge Engineering* 73, 43–57 (2012)
4. Hu, M., Lim, E.-P., Sun, A., Lauw, H.W., Vuong, B.-Q.: Measuring Article Quality in Wikipedia: Models and Evaluation. In: *CIKM 2007*, pp. 243–252. ACM Press, New York (2007)
5. Li, W.: R-calculus: An Inference System for Belief Revision. *The Computer Journal* 50(4), 378–390 (2007)
6. Libbrecht, P.: Notations Around the World: Census and Exploitation. In: Autexier, S., Calmet, J., Delahaye, D., Ion, P.D.F., Rideau, L., Rioboo, R., Sexton, A.P. (eds.) *MKM 2010. LNCS (LNAI)*, vol. 6167, pp. 398–410. Springer, Heidelberg (2010)
7. Richardson, M., Domingos, P.: Building Large Knowledge Bases by Mass Collaboration. In: *K-CAP 2003*, pp. 129–137. ACM Press, New York (2003)

Project Presentation: Algorithmic Structuring and Compression of Proofs (ASCOP)

Stefan Hetzl

Institute of Discrete Mathematics and Geometry
Vienna University of Technology
Wiedner Hauptstraße 8-10, A-1040 Vienna, Austria
hetzl@logic.at

Abstract. Computer-generated proofs are typically analytic, i.e. they essentially consist only of formulas which are present in the theorem that is shown. In contrast, mathematical proofs written by humans almost never are: they are highly structured due to the use of lemmas.

The ASCOP-project aims at developing algorithms and software which structure and abbreviate analytic proofs by computing useful lemmas. These algorithms will be based on recent groundbreaking results establishing a new connection between proof theory and formal language theory. This connection allows the application of efficient algorithms based on formal grammars to structure and compress proofs.

1 Introduction

Proofs are the most important carriers of mathematical knowledge. Logic has endowed us with formal languages for proofs which make them amenable to algorithmic treatment. From the early days of automated deduction to the current state of the art in automated and interactive theorem proving we have witnessed a huge increase in the ability of computers to search for, formalise and work with proofs. Due to the continuing formalisation of computer science (e.g. in areas such as hardware and software verification) the importance of formal proofs will grow further.

Formal proofs which are generated automatically are usually difficult or even impossible to understand for a human reader. This is due to several reasons: one is a potentially extreme length as in the well-known cases of the four colour theorem or the Kepler conjecture. But one need not go that far to make this point, a quick glance at the output of most of the current automated theorem provers may very well suffice to demonstrate this difficulty. In such cases, where mere size is not problematic, one faces logical issues such as the use of deduction formats more suited for finding than for representing proofs as well as engineering issues such as user interfaces.

Below all these aspects however is lurking a reason of a much more fundamental nature: computer-generated proofs are *analytic*, i.e. they essentially only contain such formulas which are already present in the theorem that is shown.

In contrast, human-generated mathematical proofs almost never are; in a well-structured proof the final result is usually derived from lemmas. Indeed, the computer-generated part of the proof of the four colour theorem as well as that of the the Kepler conjecture is – from the logical point of view – essentially the verification of a huge case distinction by calculations, a typical form of an analytic proof. With increasing automation in many areas, the share of such proofs can be expected to grow, another recent example being the solution to the Sudoku Minimum Number of Clues problem [15].

Such inscrutable analytic proofs *do* carry mathematical knowledge, after all they show that the theorem is true. However they carry this knowledge only in an implicit form which renders it inaccessible (to a human reader). The aim of the ASCOP-project is to develop methods and software which makes this knowledge accessible by making it explicit in the form of new lemmas.

2 Theoretical Foundations

Since the very beginning of structural proof theory, marked by the seminal work [6], it is well understood that arbitrary proofs can be transformed into analytic proofs and how to do it. This process: cut-elimination, has been the central axis of the development of proof theory in the 20th century (the inference rule *cut* formalises the use of a lemma in a proof). The approach of the ASCOP-project is to develop algorithms which *reverse* this process: starting from an analytic proof (e.g. one that has been generated algorithmically) the task is to transform it into a shorter and more structured proof of the same theorem by the introduction of cuts which – on the mathematical level – represent lemmas. An algorithmic reversal of cut-elimination is rendered possible by recent groundbreaking results (like [10,8] and [12], see also [9]) that establish a new connection between proof theory and formal language theory.

For explaining this connection, let us first consider one of the most most fundamental results about first-order logic: Herbrand's theorem [7]. In its simplest form it states that $\exists x A$, for a quantifier-free formula A , is valid iff there are terms t_1, \dots, t_n s.t. $\bigvee_{i=1}^n A[x \setminus t_i]$ is a propositional tautology. A disjunction of instances of a formula which is a tautology is therefore also often called *Herbrand-disjunction*. This theorem can be greatly generalised (see e.g. [16]), for expository purposes we stick to formulas of the form $\exists x A$ here. A Herbrand-disjunction corresponds to a cut-free proof in the sense that $\exists x A$ has a cut-free proof with n quantifier inferences iff it has a Herbrand-disjunction with n disjuncts. We write $H(\pi)$ for the set of disjuncts of the Herbrand-disjunction induced by the proof π .

It is well-known that cut-elimination may increase the length of proofs considerably, e.g. in first-order logic the growth rate is 2_n where $2_0 = 1$ and $2_{i+1} = 2^{2^i}$. Now, if a large Herbrand-disjunction arose from eliminating the cuts of a small proof, then this Herbrand-disjunction must necessarily contain a certain amount of regularity / redundancy because it has a short description: the original proof with cuts. While this observation is obvious, the question what that redundancy is and how it can be characterised and detected is much less so.

This question has recently been answered in [10,8] for the case of proofs with Σ_1 -cuts, i.e. proofs whose cut formulas have a prenex normal form $\exists x B$ where B is quantifier-free (note that a cut on a formula $\forall x B$ can easily be transformed to a Σ_1 -cut by adding a negation and switching the left and right subproofs). In [10,8] it is shown that a Herbrand-disjunction that arose from a proof π with Σ_1 -cuts can be written as the language of a **totally rigid acyclic tree grammar** that has the size of π . Rigid tree languages have been introduced in [13] with applications in verification in mind (e.g. of cryptographic protocols as in [14]). A rigid tree grammar differs from a regular tree grammar (see e.g. [5]) in that it allows certain equality constraints. Totally rigid acyclic tree grammars are a subclass of them, see [10,8] for details. Such results that describe the structure of Herbrand-disjunctions depending on the class of proofs with cut from which they originate will be called *structure theorems* in the sequel.

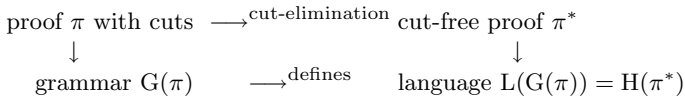


Fig. 1. Proofs and Tree Grammars

What has thus been obtained is a correspondence as depicted in Figure 1: on the level of Herbrand-disjunctions, cut-elimination is nothing but the computation of the language of a totally rigid acyclic tree grammar. Consequently this structure theorem tells us what we have to look for in a Herbrand-disjunction if we want to abbreviate it using Σ_1 -cuts: we have to **write it as the language of a totally rigid acyclic tree grammar!**

These results suggest the following systematic approach to the design of proof compression algorithms: a first theoretical step consists in proving a structure theorem for a particular class of proofs with cut. The proof compression algorithm is then designed to start from a Herbrand-disjunction H and proceed in two phases:

- First, a grammar that represents H is computed. This is a pure term problem consisting of finding a minimal (w.r.t. the number of productions) grammar for a given finite language (i.e. a trivial grammar). This problem is closely related to automata minimisation, one of the most standard problems in formal language theory.
- Secondly, cut formulas that realise this grammar in the form of a proof with cuts are computed. In the case of a single Σ_1 -cut there is always a canonical solution which is computable in linear time [11]. This property carries over to an arbitrary number of Σ_1 -cuts and – a priori – there is no reason to assume a different behaviour in the general case.

Furthermore, one obtains a completeness result of the following form: if there is a proof with cuts that leads to H via cut-elimination, the above algorithm finds

it (note the contrast to the undecidability of k/l -compressibility [3]). Therefore one also obtains a maximal compression: the algorithm finds the proof with the smallest grammar that leads to a given cut-free proof.

A first proof-of-concept algorithm realising this approach for the class of proofs having a single Σ_1 -cut is presented in [11].

3 Aims of the ASCOP-Project

The purpose of the ASCOP-project is to fully exploit the potential of this approach to structuring and compression of proofs. On the theoretical side, our main aim is to extend the classes of lemmas that can be computed beyond those in [11]. Preliminary investigations show that this extension is rather straightforward as long as the lemmas do not contain quantifier alternations. To treat those, an extension of the theoretical results of [10,8] is necessary first. As a bridge to practical applications it will also be useful to generalise these algorithms to work modulo simple theories such as equality for uninterpreted function symbols or linear arithmetic.

We will implement these proof compression algorithms based on the GAPT-project [1]. GAPT (Generic Architecture for Proofs) is a general framework for proof-theoretic algorithms implemented in Scala. Its primary application is to serve as a basis for the CERES-system [4], a system for the analysis of formalised mathematical proofs based on resolution provers. As an appropriate frame for these algorithms we envisage an implementation that allows to use the output of a resolution theorem prover as input and to compute a sequent calculus proof with cuts of the same theorem. Frequently, the user will primarily be interested in the computed lemmas, viewing the complete proof being only an option for a more detailed analysis. The existing graphical user interface of GAPT provides an adequate basis for a sufficiently flexible user interaction. As a large-scale test of our algorithms we plan to apply them as post-processing step to the output of standard resolution provers on the TPTP library [17], as in [18].

The ASCOP-project envisages a varied range of applications. In the short term we expect the system to be useful for improving the readability of the output of automated theorem provers. We furthermore expect these simplification and compression capabilities to be useful for the integration of automated provers in proof assistants (such as sledgehammer in Isabelle [2]) as they allow to break up automatically generated proofs into smaller pieces (thus facilitating their replay by Isabelle's trusted resolution prover metis). In the long term we hope that these methods have the potential to compute mathematically meaningful information from large and inscrutable analytic proofs such as that of the four colour theorem, the Kepler conjecture, the Sudoku clues proof and other similar proofs to be expected to surface in the future.

The reader interested in following the progress of the ASCOP-project is invited to consult its website at <http://www.logic.at/people/hetzl/ascop/>.

References

1. Generic Architecture for Proofs (GAPT), <http://code.google.com/p/gapt/>
2. Sledgehammer, <http://www.cl.cam.ac.uk/research/hvg/Isabelle/sledgehammer.html>
3. Baaz, M., Zach, R.: Algorithmic Structuring of Cut-free Proofs. In: Martini, S., Börger, E., Kleine Büning, H., Jäger, G., Richter, M.M. (eds.) CSL 1992. LNCS, vol. 702, pp. 29–42. Springer, Heidelberg (1993)
4. Dunchev, T., Leitsch, A., Libal, T., Weller, D., Woltzenlogel Paleo, B.: System Description: The Proof Transformation System CERES. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 427–433. Springer, Heidelberg (2010)
5. Gécseg, F., Steinby, M.: Tree Languages. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages. Beyond Words, vol. 3, pp. 1–68. Springer, Heidelberg (1997)
6. Gentzen, G.: Untersuchungen über das logische Schließen I. Mathematische Zeitschrift 39(2), 176–210 (1934)
7. Herbrand, J.: Recherches sur la théorie de la démonstration. Ph.D. thesis, Université de Paris (1930)
8. Hetzl, S.: Proofs as Tree Languages, submitted, preprint, <http://hal.archives-ouvertes.fr/hal-00613713/>
9. Hetzl, S.: On the form of witness terms. Archive for Mathematical Logic 49(5), 529–554 (2010)
10. Hetzl, S.: Applying Tree Languages in Proof Theory. In: Dediu, A.H., Martín-Vide, C. (eds.) LATA 2012. LNCS, vol. 7183, pp. 301–312. Springer, Heidelberg (2012)
11. Hetzl, S., Leitsch, A., Weller, D.: Towards Algorithmic Cut-Introduction. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18 2012. LNCS, vol. 7180, pp. 228–242. Springer, Heidelberg (2012)
12. Hetzl, S., Straßburger, L.: Herbrand-Confluence for Cut-Elimination in Classical First-Order Logic (submitted)
13. Jacquemard, F., Klay, F., Vacher, C.: Rigid Tree Automata. In: Dediu, A.H., Ionescu, A.M., Martín-Vide, C. (eds.) LATA 2009. LNCS, vol. 5457, pp. 446–457. Springer, Heidelberg (2009)
14. Jacquemard, F., Klay, F., Vacher, C.: Rigid tree automata and applications. Information and Computation 209, 486–512 (2011)
15. McGuire, G., Tugemann, B., Civario, G.: There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem, <http://arxiv.org/abs/1201.0749>
16. Miller, D.: A Compact Representation of Proofs. Studia Logica 46(4), 347–370 (1987)
17. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. Journal of Automated Reasoning 43(4), 337–362 (2009)
18. Vyskočil, J., Stanovský, D., Urban, J.: Automated Proof Compression by Invention of New Definitions. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 447–462. Springer, Heidelberg (2010)

On Formal Specification of Maple Programs^{*}

Muhammad Taimoor Khan¹ and Wolfgang Schreiner²

¹ Doktoratskolleg Computational Mathematics

² Research Institute for Symbolic Computation

Johannes Kepler University

Linz, Austria

`muhammad.khan@dk-compmath.jku.at`,

`Wolfgang.Schreiner@risc.jku.at`

<http://www.risc.jku.at/people/mtkhan/dk10/>

Abstract. This paper is an example-based demonstration of our initial results on the formal specification of programs written in the computer algebra language *MiniMaple* (a substantial subset of Maple with slight extensions). The main goal of this work is to define a verification framework for *MiniMaple*. Formal specification of *MiniMaple* programs is rather complex task as it supports non-standard types of objects, e.g. symbols and unevaluated expressions, and additional functions and predicates, e.g. runtime type tests etc. We have used the specification language to specify various computer algebra concepts respective objects of the Maple package *DifferenceDifferential* developed at our institute.

1 Introduction

We report on a project whose goal is to design and develop a tool to find behavioral errors such as type inconsistencies and violations of method preconditions in programs written in the language of the computer algebra system Maple; for this purpose, these programs need to be annotated with the types of variables and methods contracts [8].

As a starting point, we have defined a substantial subset of the computer algebra language Maple, which we call *MiniMaple*. Since type safety is a prerequisite of program correctness, we have formalized a type system for *MiniMaple* and implemented a corresponding type checker. The type checker has been applied to the Maple package *DifferenceDifferential* [2] developed at our institute for the computation of bivariate difference-differential dimension polynomials. Furthermore, we have defined a language to formally specify the behavior of *MiniMaple* programs. As the next step, we will develop a verification calculus for *MiniMaple*. The other related technical details about the work presented in this paper are discussed in the accompanying paper [7]. For project details and related software, please visit <http://www.risc.jku.at/people/mtkhan/dk10/>.

The rest of the paper is organized as follows: in Section 2, we briefly demonstrate formal type system for *MiniMaple* by an example. In Section 3, we

^{*} The research was funded by the Austrian Science Fund (FWF): W1214-N15, project DK10.

introduce and demonstrate the specification language for *MiniMaple* by an example. Section 4 presents conclusions and future work.

2 A Type System for *MiniMaple*

MiniMaple procedure parameters, return types and corresponding local (variable) declarations needs to be (manually) type annotated. Type inference would be partially possible and is planned as a later goal. The results we derive with type checking Maple can also be applied to Mathematica, as Mathematica has almost the same kinds of runtime objects as Maple.

Listing 1 gives an example of a *MiniMaple* program which we will use in the following section for the discussion of type checking respective formal specification. Also the type information produced by the type system is shown by the mapping π of program variables to types. For other related technical details of the type system, please see [4].

```

1. status:=0;
2. prod := proc(l::list(Or(integer,float))::[integer,float];
3.     #  $\pi = \{l:\text{list}(\text{Or}(\text{integer},\text{float}))\}$ 
4.     global status;
5.     local i, x::Or(integer,float), si::integer:=1, sf::float:=1.0;
6.     #  $\pi = \{\dots, i:\text{symbol}, x:\text{Or}(\text{integer},\text{float}),\dots, \text{status}:\text{anything}\}$ 
7.     for i from 1 by 1 to nops(l) do
8.         x:=l[i]; status:=i;
9.         #  $\pi = \{\dots, i:\text{integer}, \dots, \text{status}:\text{integer}\}$ 
10.        if type(x,integer) then
11.            #  $\pi = \{\dots, i:\text{integer}, x:\text{integer}, si:\text{integer}, \dots, \text{status}:\text{integer}\}$ 
12.            if (x = 0) then
13.                return [si,sf];
14.            end if;
15.            si:=si*x;
16.        elif type(x,float) then
17.            #  $\pi = \{\dots, i:\text{integer}, x:\text{float}, \dots, sf:\text{float}, \text{status}:\text{integer}\}$ 
18.            if (x < 0.5) then
19.                return [si,sf];
20.            end if;
21.            sf:=sf*x;
22.        end if;
23.        #  $\pi = \{\dots, i:\text{integer}, x:\text{Or}(\text{integer},\text{float}),\dots, \text{status}:\text{integer}\}$ 
24.    end do;
25.    #  $\pi = \{\dots, i:\text{symbol}, x:\text{Or}(\text{integer},\text{float}),\dots, \text{status}:\text{anything}\}$ 
26.    status:=1;
27.    #  $\pi = \{\dots, i:\text{symbol}, x:\text{Or}(\text{integer},\text{float}),\dots, \text{status}:\text{integer}\}$ 
28.    return [si,sf];
29. end proc;
30. result := prod([1, 8.54, 34.4, 6, 8.1, 10, 12, 5.4]);

```

Listing 1. The example *MiniMaple* procedure type-checked

The following problems arise from type checking *MiniMaple* programs:

- Global variables (declarations) can not be type annotated; therefore values of arbitrary types can be assigned to global variables in Maple. Therefore we introduce *global* and *local* contexts to handle the different semantics of the variables inside and outside of the body of a procedure respectively loop.
 - In a *global* context new variables may be introduced by assignments and the types of variables may change arbitrarily by assignments.
 - In a *local* context variables can only be introduced by declarations. The types of variables can only be *specialized* i.e. the new value of a variable should be a sub-type of the declared variable type. The sub-typing relation is observed while specializing the types of variables.
- A predicate **type**(E, T) (which is true if the value of expression E has type T) may direct the control flow of a program. If this predicate is used in a conditional, then different branches of the conditional may have different type information for the same variable. We keep track of the type information introduced by the different type tests from different branches to adequately reason about the possible types of a variable. For instance, if a variable x has type $\text{Or}(\text{integer}, \text{float})$, in a conditional statement where the "if" branch is guarded by a test $\text{type}(x, \text{integer})$, in the "else" branch x has automatically type float . This automatic type inferencing only applies if an identifier has a union type. A warning is generated, if a test is redundant (always yields true or false).

The type checker has been applied to the Maple package *DifferenceDifferential* [2]. No crucial typing errors have been found but some bad code parts have been identified that can cause problems, e.g., variables that are declared but not used (and therefore cannot be type checked) and variables that have duplicate global and local declarations.

3 A Specification Language for *MiniMaple*

Based on the type system presented in the previous section, we have developed a formal specification language for *MiniMaple*. This language is a logical formula language which is based on Maple notations but extended by new concepts. The formula language supports various forms of quantifiers, logical quantifiers (**exists** and **forall**), numerical quantifiers (**add**, **mul**, **min** and **max**) and sequential quantifier (**seq**) representing truth values, numeric values and sequence of values respectively. We have extended the corresponding Maple syntax, e.g., logical quantifiers use typed variables and numerical quantifiers are equipped with logical conditions that filter values from the specified variable range.

Also the language allows to formally specify the behavior of procedures by pre- and post-conditions and other constraints; it also supports loop specifications and assertions. In contrast to specification languages such as Java Modeling Language [3], abstract data types can be introduced to specify abstract concepts and notions from computer algebra.

```

(*@
requires true;
global status;
ensures
  (status = -1 and RESULT[1] = mul(e, e in l, type(e,integer))
  and RESULT[2] = mul(e, e in l, type(e,float))
  and forall(i::integer, 1<=i and i<=nops(l) and type(l[i],integer) implies l[i]<>0)
  and forall(i::integer, 1<=i and i<=nops(l) and type(l[i],float) implies l[i]>=0.5))
  or
  (1<=status and status<=nops(l) and RESULT[1] = mul(l[i], i=1..status-1, type(l[i],integer))
  and RESULT[2] = mul(l[i], i=1..status-1, type(l[i],float))
  and ((type(l[status],integer) and l[status]=0) or (type(l[status],float) and l[status]<0.5))
  and forall(i::integer, 1<=i and i<status and type(l[i],integer) implies l[i]<>0)
  and forall(i::integer, 1<=i and i<status and type(l[i],float) implies l[i]>=0.5));
@*)
proc(l::list(Or(integer,float))):[integer,float]; ... end proc;

```

Listing 2. The example *MiniMaple* procedure formally specified

Listing 2 gives a formal specification of the example procedure introduced in Section 2. The procedure has no pre-condition as shown in the **requires** clause; the **global** clause says that a global variable *status* can be modified by the body of the procedure. The normal behavior of the procedure is specified in the **ensures** clause. The post condition specifies that, if the complete list is processed then we get the result as the product of all integers and floats in the list but if procedure terminates pre-maturely then we only get the product of integers and floats till the value of variable *status* (index of the input list). For the complete syntax and other details of the formal specification language see [6]. To test the specification language, we have formally specified some parts of the Maple package *DifferenceDifferential* [2] developed at our institute as the main test for the specification language.

4 Conclusions

We may use the specification language sketched in this short paper to generate executable assertions that are embedded in *MiniMaple* programs and check at runtime the validity of pre/post conditions. Our main goal, however, is to use the specification language to verify the correctness of *MiniMaple* annotated programs by static analysis, in particular to detect violations of methods preconditions. For this purpose, based on the results of a prior investigation, we intend to use the verification framework Why3 [1] to implement the verification calculus for *MiniMaple*, i.e., to translate *MiniMaple* into the intermediate language of Why3 and to apply its verification condition generator to generate verification conditions and prove their correctness with various back-end provers. Since the verification calculus must be sound, we have defined a formal semantics of *MiniMaple* [5] such that the correctness of the transformation can be shown.

References

1. Bobot, F., Filliâtre, J.-C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, Wrocław, Poland (August 2011)
2. Dönch, C.: Bivariate Difference-Differential Dimension Polynomials and Their Computation in Maple. Technical report, Research Institute for Symbolic Computation, Johannes Kepler University, Linz (2009)
3. Leavens, G.T., Cheon, Y.: Design by Contract with JML. A Tutorial (2006), <ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf>
4. Khan, M.T.: A Type Checker for MiniMaple. RISC Technical Report 11-05, also DK Technical Report 2011-05, Research Institute for Symbolic Computation, Johannes Kepler University, Linz (2011)
5. Khan, M.T.: Formal Semantics of MiniMaple. DK Technical Report 2012-01, Research Institute for Symbolic Computation, Johannes Kepler University, Linz (January 2012)
6. Khan, M.T., Schreiner, W.: Towards a Behavioral Analysis of Computer Algebra Programs (Extended Abstract). In: Petterson, P., Seceleanu, C. (eds.) Proceedings of the 23rd Nordic Workshop on Programming Theory (NWPT 2011), Vasteras, Sweden, pp. 42–44 (October 2011)
7. Khan, M.T., Schreiner, W.: Towards the Formal Specification and Verification of Maple Programs. In: Jeuring, J., et al. (eds.) CICM 2012. LNCS (LNAI), vol. 7362, pp. 231–247. Springer, Heidelberg (2012)
8. Meyer, B.: Applying Design by Contract. *Computer* 25, 40–51 (1992)

The PLANETARY Project: Towards eMath3.0

Michael Kohlhase*

Computer Science,
Jacobs University Bremen, Germany
<http://planetary.mathweb.org>

Abstract. The PLANETARY project develops a general framework – the PLANETARY system – for social semantic portals that support users in interacting with STEM (Science/Technology/Engineering/Mathematics) documents. Developed from an initial attempt to replace the aging portal of PlanetMath.org with a mashup of existing MKM technologies, the PLANETARY system is now in a state, where it can serve as a basis for various eMath3.0 portals, ranging from eLearning systems over scientific archives to semantic help systems.

The PLANETARY project aims at developing a general framework – the PLANETARY system – for social semantic portals that support users in interacting with STEM documents. It is carried by enthusiasts from Jacobs University and The Open University.

Main Concepts and Project Genesis

Work on the PLANETARY system was triggered in August 2010 by the realization that the KWARC group at Jacobs University had developed semantic counterparts of much of the components underlying the PlanetMath portal [Pla]. PlanetMath.org is an online community that creates and manages an encyclopedia of mathematical concepts; hundreds of regular contributors have published about 8500 encyclopedia entries called articles. PlanetMath was founded in 2000; even before Wikipedia, and is thus one of the first Web2.0 systems. The Noösphere system [Noo] underlying the portal – essentially a L^AT_EX-based Wiki implemented in Perl – is showing its age and becoming hard to manage. We felt that extending Planetmath to an eMath3.0 system – a social semantic web platform for Mathematics – via MKM technologies, might breathe additional life into the PlanetMath community and at the same time serve as a showcase of MKM technologies into the mathematics community.

The pre-existing MKM components that can be combined to form a semantic counterpart of Noösphere are (see also Figure [I]):

1. TNTBase [Tnt] for web-enabled, versioned storage

* For the PLANETARY Group.

2. The LaTeXML daemon [GSK11] for transforming T_EX/L^AT_EX document fragments to HTML5¹
3. sT_EX [STeX], a semantic variant of L^AT_EX that can be transformed to OM-
Doc [Koh06] and further to semantically annotated HTML5 [KMR08]
4. JOBAD [JOBAD], a JavaScript API embedding semantic services into Web documents

The only missing piece was a front-end that integrated them, added user and permissions management, and added discussion fora (an essential feature of PlanetMath). We found this component in the open source Vanilla Forums system [Van], that could easily be extended by wrapping the MKM components into Vanilla plugins. Another plugin that had to be added to the mix for PlanetMath feature parity was a system for metadata management and visualization: PLANETARY exports metadata to an RDF triple store (here an instance of the Openlink Virtuoso system [Olv]) and integrates custom SPARQL queries into the user interface e.g. to allow access to PlanetMath articles via the MSC2010 classification [Msc].

Already the proof-of-concept implementation in Fall 2010 made it clear that this combination of MKM technologies could be much more useful than for just re-implementing PlanetMath.

Framework for Semantic Publishing and Active Documents

The PLANETARY system has been generalized into a comprehensive eMath3.0 framework for semantic publishing and knowledge management, which has been instantiated prototypically in a variety of settings to validate the framework and support communities. The portals realized in the PLANETARY system range from eLearning systems over scientific archives to semantic help systems. All share the common basic architecture (see Figure 1), which integrates the components discussed above as web services into a central *Container Management System* (CMS) that mediates all user interaction. Note that in this MKM-centric architecture, we greatly extend the role of the *content management subsystem* (denoted by the dotted box in Figure 1). The CMS (initially Vanilla Forums, later Drupal) supplies management and interaction at the “container level”, i.e., without ever looking into the documents it manages (hence the somewhat non-standard name). The management of *structured document*

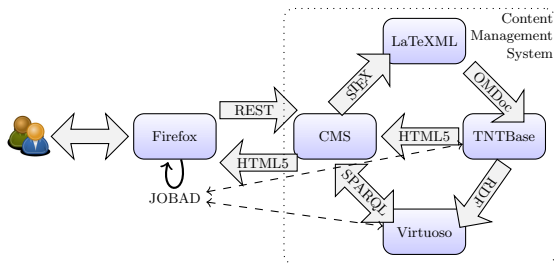


Fig. 1. Architecture of the PLANETARY system

¹ We use HTML5 as it integrates HTML for document layout with MathML for formula presentation, SVG for diagrams, and RDFa for document-embedded metadata and is supported by the major browsers.

content is split between TNTBase and the RDF triple store in PLANETARY, since they can perform semantic services.

Active Documents by Example

Note that the level of semantic interaction afforded by the PLANETARY system depends on the depth of semantic annotations in the documents, and thus on different instances of the PLANETARY system: They range from simple folding and localized commenting services in a front-end system for the Cornell ePrint arXiv to a front-end system which features in-place type reconstruction and elision of arguments and brackets for the fully formal LATIN atlas Cod+11.

We start with the former and work our way to more semantics. In all cases, services are accessible locally for objects with (fine-grained) semantic annotations – e.g. a subterm of a formula – via a special context menu menu of icons centered around the object. The icon menu has one entry per service available in the current context.

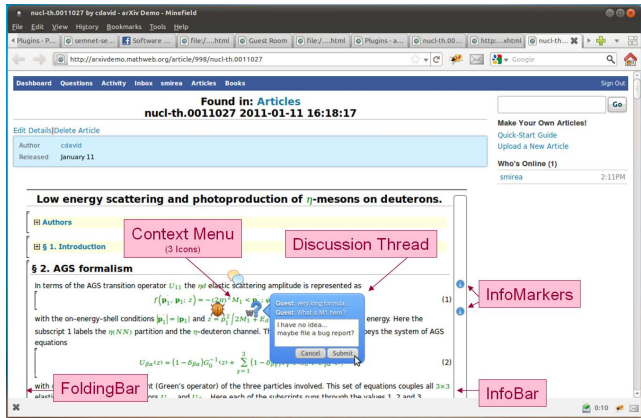


Fig. 2. Interacting with an arXiv article

For instance, the question mark icon triggers the discussion service supports localized discussion threads for reporting problems or asking questions about the selected object. The **InfoBar** on the right of Figure 2 is a secondary device that visualizes state information for the objects in the respective line of the paper, e.g. the availability of questions or discussions, which can be accessed by the icon menu.

$$s = s_i + v_i \Delta t + \frac{1}{2} a_i (\Delta t)^2$$

$$s = s_i + \dots$$

The **FoldingBar** in Figure 2, similar to source code IDEs, enables folding document structures, and the **InfoBar** icons on the right indicate the availability of local discussions. In the image on the left, we selected a subterm and requested to fold it, i.e. to simplify its display by replacing it with an ellipsis.

The richer semantic markup of OMDoc-based representations of lecture materials and the Logic Atlas collections enable services that utilize logical and functional structures – reflected by a different icon menu. Figure 3 demonstrates looking up a definition and exploring the prerequisites of a concept. The definition lookup service obtains the URI of a symbol from the annotation of a formula and queries the server for the corresponding definition. The server-side part of the prerequisite navigation service obtains the transitive closure of all dependencies of a given item and returns them as an annotated SVG graph.

maps-to is applied

$f \subseteq X \times Y$, is called a **partial function**, iff for all $x \in X$ there is at

Definition Lookup Results

DEFINITION:

Cartesian product:
 $A \times B := \{(a, b) \mid a \in A \wedge b \in B\}$, call
 (a, b) pair.

Prerequisites Graph for ./slides/dmath/en/sets-operations.tex

THIS

sets-operations

sets-introduction

mathdef-definitions

sets

mathdef

highschool

sequences

relations

sets

setsmat2

setsmat1

mathdef, highschool, sequences

Mathematics uses a very effective technique for dealing with conceptual complexity. It usually starts out with discussing simple, basic objects and their properties. These simple objects can be combined to form complex, compound ones. Then a definition is given a compound object a new name, so that it can be used like a basic one. In particular, the newly defined object can be used to form compound objects, leading to more and more complex objects that can be described succinctly. In this new mathematics increasingly research is conducted by adding new layers of definitions onto very simple and basic beginnings. We will now discuss four definition schemes that will occur over and over in this course.

Find: f is a Next Previous Highlight all Match case

Fig. 3. Definition Lookup and Prerequisites Navigation

Current and Future Work

In June 2011, the PLANETARY system became one of the finalist systems in the Elsevier Executable Papers Challenge [Koh+11]. But the development push to reach this milestone also revealed crucial shortcomings of the CMS at the heart of the PLANETARY system, and the system was ported to Drupal whose container model and editing facilities are more suitable. Unfortunately, work on the port, on improving the subsystems, and data conversion issues have delayed any deployment of production systems based on PLANETARY.

Currently, the work in the PLANETARY project focuses on four PLANETARY-based systems:

- finishing a production-ready PLANETARY instance of PlanetMath, see <http://alpha.planetmath.org>
- developing a Web2.0 frontend with lightweight semantic features for <http://arxiv.org>, an archive of over 700 000 scientific documents. Particular care will be placed on extracting functional semantics from give L^AT_EX documents and using this in formula search, see <http://arxivdemo.mathweb.org>
- re-establishing the separate compilation and linking functionality for modular semantic publishing (see [Dav+11b]) in the eLearning3.0 System PantaRhei used in teaching CS courses at Jacobs University, see <http://panta.kwarc.info> and
- integrating PLANETARY as a knowledge provider in semantic allies; see [Dav+12].

Note that all the PLANETARY instances referenced in the URIs are under active research, so your experience may vary.

References

- [Cod+11] Codescu, M., Horozal, F., Kohlhase, M., Mossakowski, T., Rabe, F.: Project Abstract: Logic atlas and integrator (LATIN). In: Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.) MKM 2011 and Calculemus 2011. LNCS(LNAI), vol. 6824, pp. 289–291. Springer, Heidelberg (2011)
- [Dav+11a] Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.): MKM 2011 and Calculemus 2011. LNCS(LNAI), vol. 6824. Springer, Heidelberg (2011)
- [Dav+11b] David, C., et al.: A Framework for Modular Semantic Publishing with Separate Compilation and Dynamic Linking. In: Castro, A.G., et al. (eds.) 1st Workshop on Semantic Publication (SePublica), Aachen. CEUR Workshop Proceedings, vol. 721 (2011), <https://svn.mathweb.org/repos/planetary/doc/sepublica11/paper.pdf>
- [Dav+12] David, C., et al.: SALLY: A Framework for Semantic Allies. In: Intelligent Computer Mathematics. LNCS (LNAI). Springer (2012), <http://kwarc.info/kohlhase/submit/mkm12-Sally.pdf>
- [GSK11] Ginev, D., Stamerjohanns, H., Miller, B.R., Kohlhase, M.: The $\mathcal{L}^{\text{ATE}}\text{XML}$ Daemon: Editable Math on the Collaborative Web. In: Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.) MKM 2011 and Calculemus 2011. LNCS(LNAI), vol. 6824, pp. 292–294. Springer, Heidelberg (2011), <https://svn.kwarc.info/repos/arXMLiv/doc/cicm-systems11/paper.pdf>
- [JOBAD] JOBAD Framework – JavaScript API for OMDoc-based active documents, <http://jobad.omdoc.org> (visited on February 18, 2012)
- [KMR08] Kohlhase, M., Müller, C., Rabe, F.: Notations for Living Mathematical Documents. In: Autexier, S., Campbell, J., Rubio, J., Sorge, V., Suzuki, M., Wiedijk, F. (eds.) AISC 2008, Calculemus 2008, and MKM 2008. LNCS (LNAI), vol. 5144, pp. 504–519. Springer, Heidelberg (2008), <http://omdoc.org/pubs/mkm08-notations.pdf>
- [Koh+11] Kohlhase, M., et al.: The Planetary System: Web 3.0 & Active Documents for STEM. Procedia Computer Science 4, 598–607 (2011); Sato, M., et al. (eds.) Special issue: Proceedings of the International Conference on Computational Science (ICCS). Finalist at the Executable Papers Challenge, <https://svn.mathweb.org/repos/planetary/doc/epc11/paper.pdf>, doi:10.1016/j.procs.2011.04.063
- [Koh06] Kohlhase, M.: OMDoc – An Open Markup Format for Mathematical Documents [version 1.2]. LNCS (LNAI), vol. 4180. Springer, Heidelberg (2006), <http://omdoc.org/pubs/omdoc1.2.pdf>
- [Msc] Mathematics Subject Classification MSC 2010 (2010), <http://msc2010.org> (visited on November 16, 2011)
- [Noo] Noosphere (diacritic optional) is the software that underlies the Planet-Math Website, <http://code.google.com/p/noosphere/> (visited on September 30, 2010)
- [Olv] OpenLink Software. OpenLink Universal Integration Middleware – Virtuoso Product Family, <http://virtuoso.openlinksw.com> (visited on October 22, 2009)
- [Pla] PlanetMath.org – Math for the people, by the people, <http://planetmath.org> (visited on September 08, 2011)
- [sTeX] Semantic Markup for $\mathcal{L}^{\text{ATE}}\text{X}$. Project Homepage, <http://trac.kwarc.info/sTeX/> (visited on February 22, 2011)
- [Tnt] TNTBase TRAC, <http://tntbase.org> (visited on December 16, 2009)
- [Van] Vanilla Forums, <http://vanillaforums.org> (visited on September 22, 2010)

Tentative Experiments with Ellipsis in Mizar

Artur Korniłowicz

Institute of Informatics
University of Białystok, Poland
arturk@mizar.org

Abstract. Ellipses are ubiquitous in mathematical texts. They allow writing sequences of terms or formulas in a concise way. In this paper, we show how ellipses are incorporated into the MIZAR language and how they are verified by the MIZAR proof checker.

1 Motivation

Mizar Mathematical Library (MML) is a collection of articles written in the MIZAR language and verified by the MIZAR proof checker [1][2]. Besides important theorems it contains series of some quite technical lemmas. For the purpose of this paper let us cite a sequence of lemmas

```
for n being Nat st n <= 1 holds n = 0 or n = 1;  
for n being Nat st n <= 2 holds n = 0 or n = 1 or n = 2;  
for n being Nat st n <= 3 holds n = 0 or n = 1 or n = 2 or n = 3;
```

stated in [3], which for a long time contained 13 such formulas. However, as the MML grew, other cases of the property were required, and increasingly more cases may be needed in the future. Therefore it would be worth to generalize the property and formulate it in a concise way

(*) for m,n being Nat st n <= m holds n = 0 or ... or n = m;

using the ellipsis (...) covering cases between the bounds.

In this paper we present how ellipses are incorporated into the MIZAR language and are verified by the MIZAR proof checker.

Ellipses are discussed and used in some other formal languages as well, e.g. [4], [5], [6] and [7].

2 Ellipsis

Extension of formal systems and languages by new constructions can be processed in different ways, e.g.:

- as syntactic sugar of some built-in mechanisms preprocessed at the syntactic level,
- by introducing new inference rules.

In the case of flexary formulas, each flexary conjunction $\phi(a) \wedge \dots \wedge \phi(b)$ could be treated as syntactic sugar of the formula $\forall_{i \in \mathbb{N}} : a \leq i \wedge i \leq b \Rightarrow \phi(i)$ and each flexary disjunction $\phi(a) \vee \dots \vee \phi(b)$ as syntactic sugar of the formula $\exists_{i \in \mathbb{N}} : a \leq i \wedge i \leq b \wedge \phi(i)$. Syntactic sugar makes texts more varied, but it does not enable us to replace the above mentioned sequence of lemmas with one lemma. If we expand (*) we get

```
for m,n being Nat st n <= m
  ex i being Nat st 0 <= i & i <= m & n = i;
```

that only states, in a complicated way, that natural numbers are non-negative.

To enable replacing such a sequence with one lemma, it is enough to add one inference rule proposed by Andrzej Trybulec

$$\frac{\alpha(i) \& \dots \& \alpha(j)}{\alpha(i) \& \dots \& \alpha(j)}$$

that may be used if the length of the conjunction is known, for example

$$\frac{\alpha(1) \& \dots \& \alpha(4)}{\alpha(1) \& \alpha(2) \& \alpha(3) \& \alpha(4)}$$

Please note the difference in fonts (different dots): $\& \dots \&$ is a MIZAR ellipsis, while $\& \dots \&$ is a meta MIZAR ellipsis.

The actual implementation is rather restricted:

- bounds i and j must be numerals (or 0 that technically in MIZAR is a nullary functor),
- the difference $j - i$ must be small. We put $j - i \leq 100$, but it is a subject of further experiments and discussions.

2.1 Parsing

In the MIZAR language, flexary formulas are represented by two new logical connectives $\& \dots \&$ and $\text{or} \dots \text{or}$ denoting the flexary conjunction and the flexary disjunction having priorities weaker than their non-flexary counterparts. That is, MIZAR allows connectives **not**, $\&$, $\& \dots \&$, **or**, $\text{or} \dots \text{or}$, **implies** and **iff** listed in the descending priority order¹. Formulas being arguments of flexary formulas have to be of the same form, i.e. for some α the left argument equals $\alpha(\tau_1)$ and the right argument equals $\alpha(\tau_2)$, where terms τ_1 and τ_2 have the type “natural number”. This implies that if we allow a formula with a flexary connective to be an argument of another flexary connective we get the formula of the form $\alpha(m, i) \wedge \dots \wedge \alpha(m, j) \wedge \dots \wedge \alpha(n, i) \wedge \dots \wedge \alpha(n, j)$. If such an embedding is applied k times, we get a formula containing $2^k - 1$ ellipses, where the main one is on the 2^{k-1} -th place. To improve the readability of such formulas, when arguments of flexary formulas are flexary, parentheses around arguments are obligatory.

¹ Implication and equivalence have the same priority.

2.2 Reasoning

Flexary connectives require establishing communication with non-flexary mathematics. This is realized by a particular formula (which we call an expansion) generated by each occurrence of a flexary connective. In the case of a flexary conjunction $\phi(a) \wedge \dots \wedge \phi(b)$ the formula is $\forall_{i \in \mathbb{N}} : a \leq i \wedge i \leq b \Rightarrow \phi(i)$, where the computation of the bounds a and b is based on terms occurring in formulas $\phi(a)$ and $\phi(b)$. The terms a and b represent the minimum difference between forms of terms of $\phi(a)$ and $\phi(b)$. For example, $\phi(n) \wedge \dots \wedge \phi(n+5)$ results in $a = n$ and $b = n+5$, while $\phi(n+0) \wedge \dots \wedge \phi(n+5)$ gives $a = 0$ and $b = 5$.

Generated expansions are used by the MIZAR checker in different aspects. MIZAR proof system is based on the Jaśkowski style of natural deduction [8] (similar systems have been developed independently by F. B. Fitch [9] and K. Ono [10]). If, for some reason, it is difficult or impossible to lead reasoning on the level of flexary formulas, the MIZAR verifier allows moving to non-flexary reasonings through such expansions. Possible forms of proof sketches are:

<pre>P[a] & ... & P[b] proof let i be natural number; assume a <= i & i <= b; thus P[i]; end;</pre>	<pre>P[a] or ... or P[b] proof take i = example; thus a <= i & i <= b; thus P[i]; end;</pre>
---	--

which are compatible with generated formulas.

2.3 Verifying

The main idea behind justification of inferences including flexary formulas is to add to each flexary formula its expansion. But, what does “to add” mean? From the satisfiability point of view there are two possible answers: the expansion can be added to the original flexary formula creating a new conjunction or a new disjunction – α is satisfiable if and only if $\alpha \wedge \hat{\alpha}$ is satisfiable and α is satisfiable if and only if $\alpha \vee \hat{\alpha}$ is satisfiable. Then we expand one copy of α getting $\alpha \wedge \hat{\alpha}$ or $\alpha \vee \hat{\alpha}$, where $\hat{\alpha}$ is the expansion of α .

Because we want to strengthen the power of the checker, what we do depends on the sign of the flexary formula to be expanded, whether it is positive or negative. If it occurs positively, i.e. when α is a premise, then we expand it to $\alpha \wedge \hat{\alpha}$. If it is negative, i.e. when $\neg\alpha$ is a premise, then we expand it to $\alpha \vee \hat{\alpha}$, what by the de Morgan law results in $\neg\alpha \wedge \neg\hat{\alpha}$. In both cases we get the stronger set of premises.

3 Applications in MML

A preliminary revision of the MML has been completed. It resulted in 43 occurrences of flexary conjunction and 201 occurrences of flexary disjunction. Primary applications are in theorems on the basic properties of natural numbers, such as

$k \leq n$ implies $k = 0$ or ... or $k = n$;
 $m \leq i$ & $i \leq m+k$ implies $i = m+0$ or ... or $i = m+k$;

Other usages occur in constructions of particular objects for some fixed limits, for example describing instructions of mathematical models of computing machines or kinds of formulas in formalizations of different logical languages and systems.

Another, quite obvious, application of flexary formulas can be the theory of finite sequences, in which many formulas are of the form

for i being natural number st $1 \leq i$ & $i \leq \text{len } f$ holds $P[i]$;

which is simply the expansion of

$$P[1] \ \& \ \dots \ \& \ P[\text{len } f];$$

The question is whether all such general statements should be rewritten in terms of flexary formulas. Since property P occurs twice in the flexary form, readability of such formulations depends on the length of the property P . When P is long, the readability is decreased.

4 Conclusions and Future Work

This paper is a report of implementing flexary conjunction and flexary disjunction in the MIZAR system. Although the current implementation is still limited—for example, a more advanced version should manage to infer

$$\frac{\phi(i) \ \& \ \dots \ \& \ \phi(j) \quad \phi(j+1) \ \& \ \dots \ \& \ \phi(k)}{\phi(i) \ \& \ \dots \ \& \ \phi(k)}$$

which the current version does not—the results are promising. For example, the usage of flexary formulas in the article `DESCIP_1` stored in the MML shortened it from 4,430 to 3,360 lines (24%) and from 153,917 to 114,975 bytes (25%).

One direction in which development of flexary formulas can proceed is defining flexary terms. For example, enumerable sets $\{F(i), \dots, F(j)\}$ could be modeled as Fraenkel terms

$$\{F(k) \text{ where } k \text{ is Nat} : k = i \text{ or } \dots \text{ or } k = j\}.$$

More advanced use of flexary formulas is connected with introducing indexed variables to the MIZAR language, what will allow us to quantify sequences of variables. Works are under discussion and design.

Modules responsible for ellipsis are incorporated into MIZAR version 7.14.01. The software can be downloaded from the MIZAR homepage: <http://mizar.org>.

Acknowledgements. Special thanks go to Andrzej Trybulec for his support during software implementation and the preparation of this paper. General ideas and concepts about ellipses in MIZAR were presented by Andrzej Trybulec during his invited talk entitled “Why the linguistic superstructure is needed?” at CICM 2011 in Bertinoro.

References

1. Mizar homepage, <http://mizar.org/>
2. Grabowski, A., Kornilowicz, A., Naumowicz, A.: Mizar in a nutshell. *Journal of Formalized Reasoning*, Special Issue: User Tutorials I 3(2), 153–245 (2010)
3. Bancerek, G.: The fundamental properties of natural numbers. *Formalized Mathematics* 1(1), 41–46 (1990)
4. Horozal, F., Kohlhase, M., Rabe, F.: Extending OpenMath with Sequences. In: Asperti, A., Davenport, J., Farmer, W., Rabe, F., Urban, J. (eds.) *Intelligent Computer Mathematics, Work-in-Progress Proceedings, Volume UBLCS-2011-04 of Technical Report*, University of Bologna, pp. 58–72 (2011)
5. Bundy, A., Richardson, J.: Proofs About Lists Using Ellipsis. In: Ganzinger, H., McAllester, D., Voronkov, A. (eds.) *LPAR 1999. LNCS (LNAI)*, vol. 1705, pp. 1–12. Springer, Heidelberg (1999)
6. Sexton, A.P., Sorge, V.: Abstract matrices in symbolic computation. In: Dumas, J.G. (ed.) *International Symposium on Symbolic and Algebraic Computation (IS-SAC)*, Genova, Italy, pp. 318–325. ACM Press (July 2006)
7. Łukaszewicz, L.: Triple dots in a formal language. *Journal of Automated Reasoning* 22, 223–239 (1999)
8. Jaśkowski, S.: On the rules of suppositions in formal logic. *Studia logica*. Nakładem Seminarjum Filozoficznego Wydziału Matematyczno-Przyrodniczego Uniwersytetu Warszawskiego (1934)
9. Fitch, F.B.: *Symbolic logic, an introduction*. Ronald Press Co., New York (1952)
10. Ono, K.: On a practical way of describing formal deductions. *Nagoya Mathematical Journal* 21, 115–121 (1962)

Reimplementing the Mathematics Subject Classification (MSC) as a Linked Open Dataset*

Christoph Lange^{1,2,3}, Patrick Ion^{4,5}, Anastasia Dimou⁵, Charalampos Bratsas⁵,
Joseph Corneli⁶, Wolfram Sperber⁷, Michael Kohlhase¹, and Ioannis Antoniou⁵

¹ Computer Science, Jacobs University Bremen, Germany

² University of Bremen, Germany

³ School of Computer Science, University of Birmingham

⁴ American Mathematical Society

⁵ Web Science, Aristotle University Thessaloniki, Greece

⁶ Knowledge Media Institute, The Open University, UK

⁷ FIZ Karlsruhe, Germany

Project page: <http://msc2010.org/mscwork/>

Abstract. The Mathematics Subject Classification (MSC) is a widely used scheme for classifying documents in mathematics by subject. Its traditional, idiosyncratic conceptualization and representation makes the scheme hard to maintain and requires custom implementations of search, query and annotation support. This limits uptake e.g. in semantic web technologies in general and the creation and exploration of connections between mathematics and related domains (e.g. science) in particular.

This paper presents the new official implementation of the MSC2010 as a Linked Open Dataset, building on SKOS (Simple Knowledge Organization System). We provide a brief overview of the dataset’s structure, its available implementations, and first applications.

Introduction

The Mathematics Subject Classification (MSC [7]) maintained by Mathematical Reviews (MR) and Zentralblatt Math (Zbl) is a widely used system for classifying mathematical documents. All major mathematical journals and digital libraries use the MSC [1], mainly as a means of structuring literature in libraries and for the purposes of retrieving information by topic. As the original format of the MSC2010 hindered its automated use and maintenance, we have reimplemented it as a machine-readable linked open dataset, which will soon be announced as the single official implementation.

* First author supported by DFG project II-[OntoSpace] of SFB/TR 8 “Spatial Cognition” and EPSRC grant “EP/J007498/1 – Formal representation and proof for cooperative games”; second author by the University of Michigan.

¹ For details about this and about most other aspects of this project presentation, we refer to a full paper published recently [6].

MSC Usage and Maintenance So Far. Previously, the right MSC class for a publication (e.g. 53A45 for “Vector and tensor analysis”) was typically chosen by consulting a human-readable document. Web forms for uploading or searching publications typically required manual input of the desired MSC classes, rather than offering assistance. The source of the MSC had been maintained in one plain T_EX file for almost 30 years. From this file, scripts produced several derived forms, including a KWIC index, a printable PDF, and HTML. Quality control of the source file with its custom macros could only be done by few experts, the derived forms did not particularly target machine processing, and the scripts that created them were once more custom, MSC-specific solutions.

Requirements for a Reimplementation. After the *content* of the MSC2010 had been settled, MR and Zbl decided to technically reimplement the source, according to the following requirements: (1) **facilitate reuse**, i.e. facilitate access, search, queries, and auto-classification; (2) **facilitate maintenance**, i.e. preserve all of the original information, use a standard format supported by existing tools, enable integration of further maintenance information into the master source (such as the changes from MSC2000 to MSC2010); (3) **enable knowledge workers and service developers to adapt and extend** the MSC for their purposes, such as connecting mathematical subjects to related fields, adding descriptive labels in further languages, without affecting the core scheme; and (4) **allow end users to explore such connections** in order to **discover new knowledge**. We chose to reimplement the MSC2010 as an RDF Linked Open Dataset, using the W3C-standardized SKOS vocabulary (Simple Knowledge Organization System [8]), which had been in use in digital libraries for several years.

Structure of the MSC/SKOS Concept Scheme

SKOS’s built-in vocabulary covers a major part of the MSC concept scheme in a straightforward way; however, for full MSC coverage, we had to extend SKOS in an MSC-specific way. We summarize the design below but refer to [6] for full technical detail.

Basics. The basic concept hierarchy was implemented as a SKOS concept scheme with 63 top-level concepts, which narrow down into 528 intermediate-level concepts, having 5606 final leaves. SKOS also supports collections of concepts besides the main hierarchy, such as “all historical topics”. The 5-character MSC class numbers (e.g. 53A45) are used as URIs of the concepts for identifying them and making their descriptions technically retrievable (see below). SKOS allows for attaching multilingual labels to concepts; so far we have English, Chinese, Italian, and Russian, each from authoritative sources, and the RDF data model allows for maintaining further, non-authoritative translations separately. Finally, SKOS supports links across concept schemes, which we have so far used for making explicit the changes from MSC1991 and MSC2000 to MSC2010.

Advanced Features. We go beyond the SKOS core, but follow established best practices, by linking MSC classes to concept schemes that have not yet fully been implemented in SKOS (the Dewey Decimal Classification). We extended the SKOS vocabulary with partitive relations, i.e. when a link to a related concept is restricted to a certain scope such as “numeric approximations” or “applications in physics”. 0.4 percent of the concepts have labels containing mathematical markup, which cannot be expressed in Unicode but requires MathML. RDF supports XML in labels, but it conflicts with multilinguality – a problem unsolved so far. Finally, we attach some information, for which we have not yet designed a dedicated representation, such as co-classification policies, as generic notes to the respective concepts.

Available Implementations

The Dataset. We generated the new SKOS master source of the MSC2010 with a Perl script that translated the old \TeX source to RDF/XML. We publish the data in four complementary ways, aiming to address a large audience of users and developers. The project page at <http://msc2010.org/mscwork/> contains links to all alternatives. For **linked data access** (i.e. directly retrieving an RDF description of each MSC class by dereferencing its URI; cf. [3]), we split the SKOS master file into one file per MSC class and serve the latter with the `application/rdf+xml` MIME type. For *querying* the dataset, we expose it through a SPARQL endpoint, i.e. a standardized interface for querying RDF. For *browsing*, we offer a wiki frontend. Finally, for application developers, we offer the whole dataset for download as self-contained files in several formats.

Easy and reliable maintenance requires a master source without redundancy, e.g. the concept hierarchy should not be represented both top-down and bottom-up. Linked data browsing, as well as other uses in large-scale applications that use plain RDF without inferencing support for performance, require a richer dataset with most practically relevant entailments expanded. We automatically create this expanded version by applying a set of first-order rules implemented in N3, covering a subset of the SKOS semantics and our MSC-specific extensions.

In Use. The new implementation of the MSC2010 is currently in use on two websites linked from the project page. Both have been developed independently using the RDF-aware Drupal 7 content management system. On the homepage of the School of Mathematics at Aristotle University Thessaloniki (AUTH), we annotated the scientific fields covered by the courses and the faculty’s research interests in terms of MSC/SKOS. The new version of the PlanetMath encyclopedia (shortly entering beta testing) is powered by the Planetary system [4]. In the old version, MSC-based navigation accounted for estimated 5–6 percent of all accesses. We have now reimplemented this functionality in a more generic way: all article metadata (including MSC classifications) are maintained within \LaTeX preambles, the articles are transformed from \LaTeX to XHTML+RDFa, from which RDF is harvested into an RDF triple store, which also holds a copy

of the MSC/SKOS dataset. This approach is more flexible than the older static MSC access: our plan is to use SPARQL queries to expose specific slices of the encyclopedia content (e.g. “all articles by my co-authors in algebraic topology”). The following listing, slightly adapted from the actual situation in PlanetMath, shows a SPARQL query that returns the subclasses of the given MSC class (which would actually be a parameterizable variable), and the number of articles classified with them²:

```

PREFIX msc: <http://msc2010.org/resources/MS/2010/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX dct: <http://purl.org/dc/terms/>
SELECT DISTINCT ?subclass ?notation ?label COUNT(?article) WHERE {
  msc:53Axx skos:narrower ?subclass . # get subclasses; then, for each subclass:
  ?subclass skos:notation ?notation ; # get the MSC class number
  skos:prefLabel ?label . # get the preferred label
  OPTIONAL { ?article dct:subject ?subclass } # get classified articles (if any)
  FILTER langMatches(lang(?label), "en") # only English labels
}
GROUP BY ?subclass ?notation ?label # grouping just needed for COUNT() to work

```

Conclusion and Future Work

We deliver the first implementation of the MSC that is easily comprehensible to machines. It does not only facilitate maintenance and development of novel services, but the rigorous conceptual modeling approach [6] also helped to uncover issues in the MSC conceptualization, which we have now fixed. We are now planning to do three things in parallel: (1) refining the dataset implementation by making the internal structures of the MSC even more explicit³ and by adopting further best practices for modeling classification schemes in SKOS [9]; (2) supporting the MKM community in building applications that make use of the MSC, using our new implementation; and (3) interlinking the MSC dataset with further mathematical and non-mathematical datasets, particularly including the OpenMath Content Dictionaries [5] and DBpedia [2].

References

- [1] ARQ – A SPARQL Processor for Jena, <http://jena.sourceforge.net/ARQ/> (visited on April 23, 2012)
- [2] DBpedia, <http://dbpedia.org> (visited on January 23, 2010)

² The number is determined using the *COUNT()* aggregation function provided by the ARQ extension to SPARQL [1].

³ So far, our implementation makes two structural aspects more explicit than the old \TeX source: the concept hierarchy, which was previously given implicitly by the numbering scheme, and the cross-references, parts of which were previously given in natural language. Our current implementation does not yet explicitly represent co-classification policies.

- [3] Heath, T., Bizer, C.: Linked Data: Evolving the Web into a Global Data Space, 1st edn. Synthesis Lectures on the Semantic Web: Theory and Technology. Morgan & Claypool, San Rafael (2011), <http://linkeddatabook.com>
- [4] Kohlhase, M., et al.: The Planetary System: Web 3.0 & Active Documents for STEM. *Procedia Computer Science* 4, 598–607 (2011); Sato, M., et al. (eds.) Special issue: Proceedings of the International Conference on Computational Science (ICCS). Finalist at the Executable Papers Challenge, <https://svn.mathweb.org/repos/planetary/doc/epc11/paper.pdf>, doi:10.1016/j.procs.2011.04.063
- [5] Lange, C.: Krextor - An Extensible Framework for Contributing Content Math to the Web of Data. In: Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.) MKM 2011 and Calculemus 2011. LNCS(LNAI), vol. 6824, pp. 304–306. Springer, Heidelberg (2011) ISBN: 978-3-642-22672-4, <http://kwarc.info/clange/pubs/krextor-system.pdf>
- [6] Lange, C., Ion, P., Dimou, A., Bratsas, C., Sperber, W., Kohlhase, M., Antoniou, I.: Bringing Mathematics to the Web of Data: The Case of the Mathematics Subject Classification. In: Simperl, E. (ed.) ESWC 2012. LNCS, vol. 7295, pp. 763–777. Springer, Heidelberg (2012), <http://kwarc.info/clange/pubs/eswc2012-mscscos>
- [7] Mathematics Subject Classification MSC 2010 (2010), <http://msc2010.org> (visited on November 16, 2011)
- [8] Miles, A., Bechhofer, S.: SKOS Simple Knowledge Organization System Reference. W3C Recommendation. World Wide Web Consortium (W3C) (August 18, 2009), <http://www.w3.org/TR/2009/REC-skos-reference-20090818/>
- [9] Panzer, M., Zeng, M.L.: Modeling Classification Systems in SKOS: Some Challenges and Best-Practice Recommendations. In: Proceedings of the International Conference on Dublin Core and Metadata Applications (Seoul). Dublin Core Metadata Initiative (2009), <http://dcpapers.dublincore.org/index.php/pubs/article/view/974/0>

The Distributed Ontology Language (DOL): Ontology Integration and Interoperability Applied to Mathematical Formalization*

Christoph Lange^{1,2}, Oliver Kutz¹, Till Mossakowski^{1,3}, and Michael Grüninger⁴

¹ Research Center on Spatial Cognition, University of Bremen

² School of Computer Science, University of Birmingham

³ DFKI GmbH Bremen

⁴ Department of Mechanical and Industrial Engineering, University of Toronto

<http://ontolog.cim3.net/cgi-bin/wiki.pl?OntoIOP>

Abstract. The Distributed Ontology Language (DOL) is currently being standardized within the OntoIOP (Ontology Integration and Interoperability) activity of ISO/TC 37/SC 3. It aims at providing a unified framework for (1) ontologies formalized in heterogeneous logics, (2) modular ontologies, (3) links between ontologies, and (4) annotation of ontologies. This paper focuses on an application of DOL's meta-theoretical features in mathematical formalization: validating relationships between ontological formalizations of mathematical concepts in COLORE (Common Logic Repository), which provide the foundation for formalizing real-world notions such as spatial and temporal relations.

1 Distributed Ontologies for Interoperability

An ontology is a formal description (in a logical language) of the concepts and relationships that are of interest to an agent (user or service) or a community of agents. Today, ontologies are applied in virtually all information-rich endeavors, for example eBusiness, eHealth, eLearning, and ambient assisted living. Ontologies facilitate semantic integration of data and services by providing a common formal model, onto which data from different sources, as well as descriptions of different services, can be mapped.

Complex applications, which involve multiple ontologies with overlapping concept spaces, also require data mapping on a higher level of abstraction, viz. between different ontologies, where it is called ontology alignment. While ontology alignment is most commonly studied for ontologies in the same logic, the different ontologies driving complex applications may also be formalized in *different* logics. Popular choices include propositional logic (e.g. in industrial requirements engineering), description logic (e.g. in biomedical applications and semantic web

* The development of DOL is supported by the German Research Foundation (DFG), Project II-[OntoSpace] of the SFB/TR 8 "Spatial Cognition"; the first author is additionally supported by EPSRC grant EP/J007498/1. The authors would like to thank the OntoIOP working group within ISO/TC 37/SC 3 for their feedback.

services), and first-order logic (required for formalizing mereology and notions of space and time, but exhibiting undecidable reasoning tasks). Our approach faces this diversity not by proposing yet another ontology language – based on a logic that would subsume all the others – but instead we *accept the diverse reality* and formulate means (on a sound and formal semantic basis) to *compare and integrate ontologies formalized in different logics*. We aim at addressing the challenge of automatically checking the coherence (e.g. consistency, conservativity, intended consequences) of ontologies and ontology-based services.

2 The Distributed Ontology Language (DOL) – Overview

An ontology in the Distributed Ontology Language (DOL) consists of modules formalized in basic ontology languages, such as OWL (based on description logic) or Common Logic (based on first-order logic with some second-order features). These modules are serialized in the existing syntaxes of these languages as to facilitate reuse of existing ontologies. DOL adds a meta-level on top, which allows for expressing heterogeneous ontologies and links between ontologies¹. Such links include (heterogeneous) imports and alignments, conservative extensions (important for the study of ontology modules), and theory interpretations (important for reusing proofs). Thus, DOL gives ontology interoperability a formal grounding and makes heterogeneous ontologies and services based on them amenable to automated verification.

DOL is currently being standardized within the OntoIOP (Ontology Integration and Interoperability) activity of ISO/TC 37/SC 3². The international working group comprises around 50 experts (around 15 active contributors so far), representing a large number of communities in ontological research and application, such as different (1) ontology languages and logics (e.g. the Common Logic and OWL), (2) conceptual and theoretical foundations (e.g. model theory), (3) technical foundations (e.g. ontology engineering methodologies and linked open data), and (4) application areas (e.g. manufacturing). For details and earlier publications, see the OntoIOP project page.

The OntoIOP/DOL standard is currently in the working draft stage and will be submitted as a committee draft (the first formal ISO standardization stage) in August 2012³. The final international standard ISO 17347 is scheduled for 2015. The standard specifies syntax, semantics, and conformance criteria:

Syntax: abstract syntax of distributed ontologies and their parts; three concrete syntaxes: a text-oriented one for humans, XML and RDF for exchange among tools and services, where RDF particularly addresses exchange on the Web.

¹ The languages that we call “basic” ontology languages here are usually limited to one logic and do not provide meta-theoretical constructs.

² TC = technical committee, SC = subcommittee.

³ The standard draft itself is not publicly available, but negotiations are under way to make the final standard document public, as has been done with the related Common Logic standard [\[2\]](#).

Semantics: (1) a *direct set-theoretical semantics* for the core of the language, extended by an *institutional and category-theoretic semantics* for advanced features such as ontology combinations (technically co-limits), where basic ontologies keep their original semantics; (2) a *translational semantics*, employing the semantics of the expressive Common Logic ontology language for all basic ontologies, taking advantage of the fact that for all basic ontology languages known so far translations to Common Logic have been specified or are known to exist⁴; (3) finally, there is the option of providing a *collapsed semantics*, where the semantics of the meta-theoretical language level provided by DOL (logically heterogeneous ontologies and links between them) is not just specified on paper in semiformal mathematical textbook style, but once more formalized in Common Logic, thus in principle allowing for machine verification of meta properties. For details about the semantics, see [6].

Conformance criteria provide for DOL’s extensibility to other basic ontology languages than those considered so far, including possible future languages. (1) A *basic ontology language* conforms with DOL if its underlying logic has a set-theoretic or, for the extended DOL features, an institutional semantics. Similar criteria apply to translations between languages. (2) A concrete syntax (*serialization*) of a basic ontology language conforms if it supports IRIs (Unicode-aware Web-scalable identifiers) for symbols and satisfies some further well-formedness criteria. (3) A *document* conforms if it is well-formed w.r.t. one of the DOL concrete syntaxes, which particularly requires explicitly mentioning all logics and translations employed. (4) An *application* essentially conforms if it is capable of processing conforming documents, and providing logical information that is implied by the formal semantics.

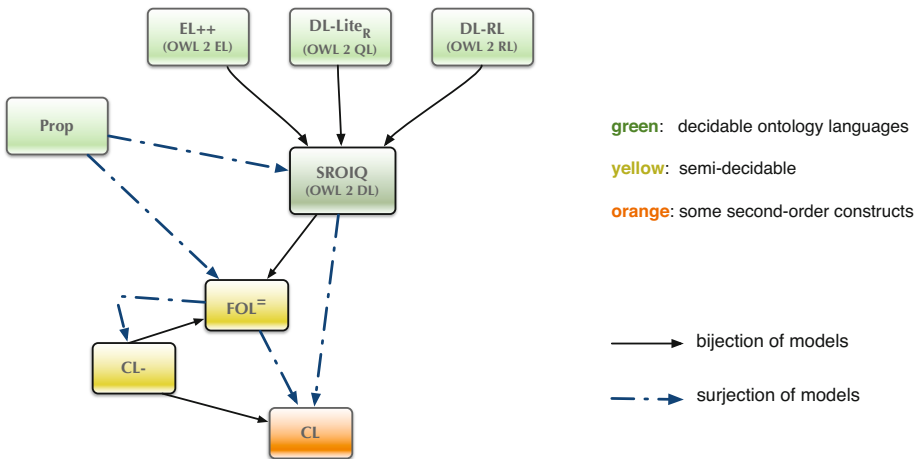


Fig. 1. A core logic translation graph for basic ontology languages

⁴ Even for higher-order logics this works, in principle, by using combinators.

Figure 1 shows some logics and translations relevant for ontologies⁵: *SROIQ*, the logic of OWL, and its sublogics corresponding to sublanguages (“profiles”) of OWL – \mathcal{EL}^{++} , DL-Lite_R, and RL – which aim at lowering the complexity of reasoning in certain relevant application domains; propositional logic; first-order logic with equality, Common Logic without second-order features, and full Common Logic. We have defined the translations between them in earlier publications [6, 5]; elaborating these definitions into annexes to the standard, which establish the conformance of these logics and translations with DOL, remains to be done.

3 Applications to Mathematical Formalization

The first application of DOL can be seen in COLORE, the Common Logic Repository, an open repository of more than 500 Common Logic ontologies. The objective of COLORE is to provide an “adequate set of generic ontologies that can be used to specify the semantics of primitive concepts”, as, for example, “any product ontology must refer to relationships from geometry and topology, and different manufacturing standards may require different ontologies for time”⁶. One of the primary applications of COLORE is to support the verification of ontologies for commonsense domains such as time, space, shape, and processes. Verification consists in proving that the ontology is equivalent to a set of core ontologies for mathematical domains such as orderings, incidence structures, graphs, and algebraic structures. COLORE comprises core ontologies that formalize algebraic structures (such as groups, fields, and vector spaces), orderings (such as partial orderings, lattices, betweenness), graphs, and incidence structures in Common Logic, and, based on these, representation theorems for generic ontologies for the above-mentioned commonsense domains.

Meta-theoretical relationships between these ontologies are of particular interest, including maps (signature morphisms), definitional extension, conservative extension, inconsistency between modules, imports, relative interpretation, faithful interpretation, and definable equivalence. DOL allows for formalizing them (as compared to the earlier approach of just writing them down as metadata), and we have started to automatically verify them using Hets (Heterogeneous Tool Set [7]). The listing below shows an example⁷ for interpreting linear orders (`linear_ordering`) as orders between time intervals that begin and end with an instant (`owltime_le`). A third ontology (`mappings/owltime2orderings`) takes care of mapping the different predicate names used by the source and the target ontology, respectively. We state that the source ontology can be interpreted in terms of the union of the target ontology and the mapping ontology in a

⁵ The logics and translations have mostly been specified as part of the Logic Atlas [1].

⁶ <http://colore.googlecode.com>

⁷ An excerpt from https://colore.googlecode.com/svn/trunk/ontologies/complex/owltime/owltime_interval/mappings/owltime_le.dol; the individual ontologies are actually stored in separate files, but here we demonstrate DOL’s ability to maintain different ontologies within one file.

model-theoretically conservative way, and that `mappings/owltime2orderings` extends `owltime_le` with definitions.

```
%prefix(                                     % prefixes for abbreviating long IRIs: this distributed ontology
: <http://code.google.com/p/colore/.../owltime/owltime_interval/mappings/owltime_le.dol#>
log: <http://purl.net/dol/logics/>           % DOL-conforming logics (Fig. 11)
ser: <http://purl.net/dol/serializations/>   % serializations, i.e. concrete syntaxes
int: <http://code.google.com/p/colore/.../owltime/owltime_interval/> % namespaces of the ontologies
ord: <http://code.google.com/p/colore/.../orderings/> % in this distributed ontology

%% The following ontologies are in the logic Common Logic, and written in the Lisp-style CLIF syntax
Logic log:CommonLogic syntax ser:CommonLogic/CLIF

ontology ord:linear_ordering =               % Here we use Common Logic's ontology import facility, but ...
  (cl-imports ord:partial_ordering) (forall (x y) (or (leq x y) (leq y x) (= x y)))

%% DOL also has a general import facility: We create the ontology of linearly ordered time intervals
ontology int:owltime_le = int:owltime_linear then int:owltime_e % that begin and end with an instant
%% ... by extending linearly ordered time intervals with intervals that begin and end with an instant

ontology int:mappings/owltime2orderings = (forall (x y) (iff (leq x y) (or (before x y) (= x y))))
  (forall (x y) (iff (lt x y) (before x y))) % map time intervals to general linear orderings

interpretation i %mcons :                   % interpreting linear orderings as time interval orders
ord:linear_ordering to {int:owltime_le and %def int:mappings/owltime2orderings}
```

DOL can also be used to specify the relationships between ontologies axiomatized in different logics. There are several cases in which there exist ontologies for the same domain, some of which are axiomatized in description logic and others in first-order logic. The best example of this is OWL-Time, which was originally proposed with an OWL axiomatization, and later extended with a first-order axiomatization [4]. (COLORE includes a modularized version of OWL-Time [3]; the listing shows some of the modules.) Using DOL, one can specify that the first-order axiomatization is a nonconservative extension of the OWL axiomatization, but that there exists a subtheory of the first-order axiomatization that is definably equivalent to the OWL axiomatization.

References

- [1] Codescu, M., Horozal, F., Kohlhase, M., Mossakowski, T., Rabe, F.: Project Abstract: Logic Atlas and Integrator (LATIN). In: Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.) *Calculemus/MKM 2011*. LNCS (LNAI), vol. 6824, pp. 289–291. Springer, Heidelberg (2011)
- [2] Common Logic (CL): a framework for a family of logic-based languages. Technical Report 24707, ISO/IEC (2007), <http://iso-commonlogic.org>
- [3] Grüninger, M.: Verification of the OWL-Time Ontology. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) *ISWC 2011, Part I*. LNCS, vol. 7031, pp. 225–240. Springer, Heidelberg (2011)
- [4] Hobbs, J., Pan, F.: An Ontology of Time for the Semantic Web. *ACM Transactions on Asian Language Processing* 3, 66–85 (2004)
- [5] Mossakowski, T., Kutz, O.: The Onto-Logical Translation Graph. In: Kutz, O., Schneider, T. (eds.) *Modular Ontologies*. IOS (2011)
- [6] Mossakowski, T., Kutz, O., Lange, C.: Three Semantics for the Core of the Distributed Ontology Language. In: FOIS (in press, 2012)
- [7] Mossakowski, T., Maeder, C., Lüttich, K.: The Heterogeneous Tool Set, HETS. In: Grumberg, O., Huth, M. (eds.) *TACAS 2007*. LNCS, vol. 4424, pp. 519–522. Springer, Heidelberg (2007)

Isabelle/jEdit – A Prover IDE within the PIDE Framework

Makarius Wenzel

Univ. Paris-Sud, Laboratoire LRI, UMR8623, Orsay, F-91405, France
CNRS, Orsay, F-91405, France
<http://www.lri.fr/~wenzel/>

1 Overview

PIDE is a general framework for document-oriented prover interaction and integration, based on a bilingual architecture that combines ML and Scala [2]. The overall aim is to connect LCF-style provers like Isabelle [5, §6] (or Coq [5, §4] or HOL [5, §1]) with sophisticated front-end technology on the JVM platform, overcoming command-line interaction at last.

The present system description specifically covers Isabelle/jEdit as part of the official release of Isabelle2011-1 (October 2011). It is a concrete Prover IDE implementation based on Isabelle/PIDE library modules (implemented in Scala) on the one hand, and the well-known text editor framework of jEdit (implemented in Java) on the other hand.

The interaction model of our Prover IDE follows the idea of continuous proof checking: the theory source text is annotated by semantic information by the prover as it becomes available incrementally. This works via an asynchronous protocol that neither blocks the editor nor stops the prover from exploiting parallelism on multi-core hardware. The jEdit GUI provides standard metaphors for augmented text editing (highlighting, squiggles, tooltips, hyperlinks etc.) that we have instrumented to render the formal content from the prover context. Further refinement of the jEdit display engine via suitable plugins and fonts approximates mathematical rendering in the text buffer, including symbols from the \TeX repertoire, and sub-/superscripts.

Isabelle/jEdit is presented here both as a usable interface for current Isabelle, and as a reference application to inspire further projects based on PIDE.

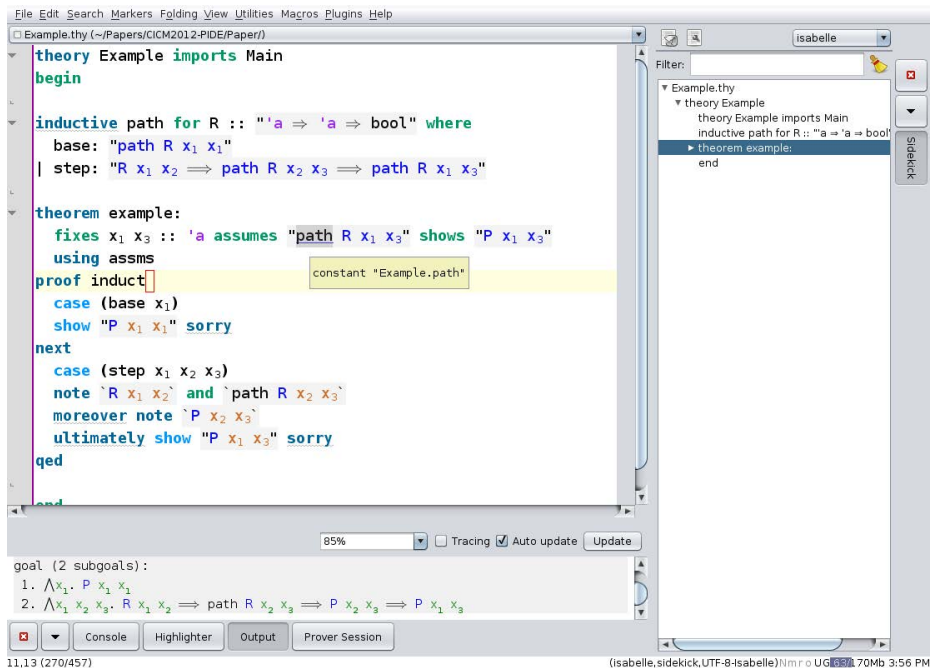
2 Using the System

The described system is part of the official release Isabelle2011-1 (October 2011). The download archives from <http://isabelle.in.tum.de/website-Isabelle2011-1/download.html> cover the three main platform families: Linux, Mac OS X, and Windows (with Cygwin). Isabelle/jEdit has already a history of about 4 years; a preliminary version is discussed in [3]. October 2011 marks the point of the first stable release of the Prover IDE; some remaining limitations are described in

its README panel. The website <http://isabelle.in.tum.de> points dynamically to the latest official release, and further improvements of Isabelle/jEdit can be anticipated with coming Isabelle distributions.

The Isabelle distribution bundles sources and multi-platform binaries, including Isabelle/jEdit. Conceptually, the Prover IDE is a *rich-client platform* with significant hard-disk foot-print, but it runs seamlessly for most users. The shell command `Isabelle2011-1/bin/isabelle jedit` opens a text editor session of jEdit which we have augmented by some plugins to communicate with the prover in the background. Source files with `.thy` extension are treated specifically: Isabelle/jEdit adds them to the formal document-model of Isabelle/PIDE, that maintains semantic information provided by the prover in the background, while the user is editing the text in the foreground.

The subsequent screenshot shows the editor view after opening a certain `Example.thy` file. The Isabelle distribution contains many other examples, e.g. `Isabelle2011-1/src/HOL/Unix/Unix.thy` where the editor will also propose to load further imported theory files.



The main text area is surrounded by *dockable windows* that are associated with jEdit plugins. For example, we provide *Output* for prover messages and traditional goal states, which is internally based on existing HTML4/CSS2 rendering on the JVM. The tree view is provided by *Sidekick*, which is an existing jEdit plugin that has been instrumented to understand some Isabelle theory structure.

The general aim of Isabelle/jEdit is to expose the specific virtues of both Isabelle and jEdit to the user, without accidental technical limitations imposed

on either system. This is in contrast to the classic Proof General / Emacs [1], where the *locked region* is essentially an intrusion of the prover command prompt into the editor; it restricts the user to a single focus inherited from TTY mode.

Having replaced the prover Read-Eval-Print-Loop by native document editing in Isabelle/PIDE, we can connect the editor more directly. The sophisticated features that qualify jEdit as “Programmer’s Text Editor”¹ are retained, and augmented by the semantic information from the prover. The underlying JVM platform is sufficiently flexible to support our requirements for this formal document-model, but instead of Java we are always using Scala [2] for our own implementations. Higher-order functional-object-oriented programming on multi-threaded JVM is far removed from untyped single-threaded Emacs-Lisp.

Physical rendering of document content draws from the standard repertoire of known IDEs for programming languages, with highlighting, squiggles, tooltips, hyperlinks etc. In the above screenshot, only the bold keywords of the Isar language use traditional syntax-highlighting in jEdit with static tables; all other coloring is based on dynamic information from the logical context of the prover.

Such annotated text regions can be explored further by using the CONTROL modifier key (or COMMAND on Mac OS X), together with mouse hovering or clicking. It reveals tooltips and hyperlinks, e.g. see `constant "Example.path"` above, and thus explains how a certain piece of source text has been interpreted.

The combination of Isabelle/jEdit and the underlying semantic document-model should help users that are accustomed to Netbeans or Eclipse to approach formal logic and formalized mathematics. Thus we hope to see new generations of users continuing the tradition of the “LCF approach” from the 1970-ies.

3 Implemented Concepts

Conceptually, the implementation consists of two main parts: (1) Isabelle/PIDE infrastructure in ML and Scala that is considered an integral part of the prover distribution, and (2) Isabelle/jEdit plugins and supporting code to assemble the main application. PIDE provides the main concepts for document-oriented interaction, and is most challenging to implement. Some aspects of previous versions are described in [3, 4], but the main issues are still unpublished. Compared to that the jEdit application is relatively small and simple: ≈ 100 Kb of Scala code.

The implemented concepts of Isabelle/jEdit in Isabelle2011-1 that are visible to end-users are as follows:

Continuous checking of source text while editing; no locking, no need to save intermediate files.

Dependency management between text buffers: each theory file corresponds to a node in the development graph of the current Isabelle session. Imports are resolved by reloading required files; edits on some node are propagated through the dependency graph as expected.

¹ <http://www.jedit.org>

Limitation: non-theory add-on files still need to be managed manually, to ensure that the prover loads the proper version.

Status overview of single text buffers and the overall prover session, with incremental update while the prover processes theories and proofs (usually in parallel on multi-core hardware).

Annotated input of source text, which is semantically decorated and physically highlighted via standard GUI metaphors.

Annotated output of prover messages, which is produced by traditional pretty-printing of the term language that is augmented by semantic markup. Rendering is delegated to HTML4/CSS on the JVM.

Limitation: no hyperlinks within the browser window yet.

Integration of Isabelle/ML into the Prover IDE: ML source inside Isar is fully annotated by the compiler, with inferred types and identifier scopes.

Integration of Isabelle/Scala into the jEdit *Console* plugin, which provides command line to access the running JVM via the Scala toplevel.

Limitation: only minimal IDE integration via terminal window.

Mathematical rendering of the source text based on Unicode characters, custom-made *IsabelleText* font with common glyphs from the $\text{T}_{\text{E}}\text{X}$ repertoire, and sub-/superscripts via extended jEdit text styles

Limitation: only 1-dimensional layout following traditional text editing, no support for 2-dimensional boxes (fractions, roots, matrices).

Completion mechanism for mathematical symbols and keywords of the formal Isar language.

Limitation: based on static tables, no connection to semantic context yet.

Regular jEdit functionality and generic plugin can be used as well. The physical representation of formal sources coincides with JVM and jEdit conventions. So copy-and-paste or hyper-search of mathematical symbols does not cause any surprises to jEdit users.

References

- [1] Aspinall, D.: Proof General: A Generic Tool for Proof Development. In: Graf, S. (ed.) TACAS 2000. LNCS, vol. 1785, pp. 38–43. Springer, Heidelberg (2000)
- [2] Odersky, M., et al.: An overview of the Scala programming language. Technical Report IC/2004/64, EPF Lausanne (2004)
- [3] Wenzel, M.: Asynchronous proof processing with Isabelle/Scala and Isabelle/jEdit. In: Coen, C.S., Aspinall, D. (eds.) User Interfaces for Theorem Provers (UITP 2010). ENTCS (July 2010); FLOC 2010 Satellite Workshop
- [4] Wenzel, M.: Isabelle as Document-Oriented Proof Assistant. In: Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.) Calculemus/MKM 2011. LNCS (LNAI), vol. 6824, pp. 244–259. Springer, Heidelberg (2011)
- [5] Wiedijk, F. (ed.): The Seventeen Provers of the World. LNCS (LNAI), vol. 3600. Springer, Heidelberg (2006)

Author Index

- Alama, Jesse 1, 427
Antoniou, Ioannis 458
Asperti, Andrea 17, 417
Aspinall, David 186
- Baker, Josef B. 422
Bourke, Timothy 32
Bradford, Russell J. 280
Bratsas, Charalampos 458
Bylinski, Czesław 427
- Carette, Jacques 202
Chen, Xiaoyu 432
Corneli, Joseph 458
- Daum, Matthias 32
Davenport, James H. 280
David, Catalin 49
de Moura, Leonardo 358
Dimou, Anastasia 458
Dos Reis, Gabriel 295
- Echenim, Mnacho 310
- Fleuriot, Jacques 371
- Grov, Gudmund 186
Grüninger, Michael 463
- Heras, Jónathan 216
Herding, Daniel 111
Hetzl, Stefan 438
Horozal, Fulya 65
Hu, Rui 81
- Iancu, Mihnea 326
Ion, Patrick 458
- Janičić, Predrag 127, 264
Jucovschi, Constantin 49, 96
- Khan, Muhammad Taimoor 231, 443
Klein, Gerwin 32
Kofler, Kevin 386
Kohlhase, Andrea 49
Kohlhase, Michael 49, 65, 342, 448, 458
Kolanski, Rafal 32
Kornilowicz, Artur 453
Kutz, Oliver 463
- Lange, Christoph 169, 458, 463
Li, Wei 432
Libbrecht, Paul 111
Luo, Jie 432
- Mamane, Lionel 1
Marić, Filip 248
Marinković, Vesna 127
Matican, Bogdan A. 342
Mazalov, Vadim 81, 402
Mossakowski, Till 463
Müller, Wolfgang 111
- Neumaier, Arnold 386
Nikolić, Mladen 264
- O'Connor, Russell 202
- Passmore, Grant Olney 358
Paulson, Lawrence C. 358
Peltier, Nicolas 310
Poza, María 216
Prodescu, Corneliu-Claudiu 342
- Rabe, Florian 65, 143, 326
Rebholz, Sandra 111
Ricciotti, Wilmer 417
Rubio, Julio 216
- Schreiner, Wolfgang 231, 443
Scott, Phil 371
Sexton, Alan P. 159, 422
Sorge, Volker 422
Sperber, Wolfram 458
- Tankink, Carst 169
Tscheulin, Felix 111
- Urban, Josef 1, 169
- Vučković, Bojan 248
- Wang, Dongming 432
Watt, Stephen M. 81, 402
Wenzel, Makarius 468
Whiteside, Iain 186
Wilson, David J. 280
- Živković, Miodrag 248