Mathematical Assistant Systems for Theory Exploration and Mathematics Education

Johannes Kepler University Linz (JKU)
Research Institute for Symbolic Computation (RISC)

Mathematical Assistant Systems for Theory Exploration and Mathematics Education

Wolfgang Windsteiger

Habilitation Thesis

Submitted in partial fulfillment of the requirements
for a *venia legendi* in *Symbolic Computation*

March 2014

Abstract

The habilitation thesis consists of the *fourteen refereed publications collected in Section II.1* of this volume. Chapter I gives an overview on the topic covered in this thesis, which ranges from the *design and implementation of mathematical assistant systems* through various *applications of mathematical assistants* to finally the *interactions between mathematical software and mathematics education*. The chapter features compact summaries of the individual publications in the thesis, fits them into the context of the topics listed above, and carefully points out their interrelations.

CONTENTS

Mathematical Assistant Systems for Theory Exploration and Mathematics Education

Wolfgang Windsteiger

I OVERVIEW ON THE TOPIC OF THE HABILITATION THESIS **1**
1 Introduction 1
2 Mathematical Assistants: Design & Implementation 4
3 Mathematical Assistants: Use in Theory Exploration & Education 12
4 Mathematical Assistants: Interplay with Education 17

II PUBLICATIONS **23**
1 Refereed Publications Making Up the Habilitation Thesis 23
2 Related Publications 227

III CURRICULUM VITAE **229**

CHAPTER I

OVERVIEW ON THE TOPIC OF THE HABILITATION THESIS

1 Introduction

There is no strict definition of what a *mathematical assistant system* is. We want to view mathematical assistant systems as the "pencil and paper of the twenty-first century". They are computer systems designed to support and facilitate *all aspects* of the work of a mathematician, i.e. *everything* that a pre-twenty-first-century mathematician would be doing using pencil and paper (and a typewriter or a word-processing program). In this context, the typical tasks of a mathematician range from the formulation of individual theorems or entire theories, through conjecturing and proving mathematical statements, performing computations, designing algorithms and executing the respective programs, to finally presenting mathematics in a visually appealing form in papers, lecture notes, or presentations. This list is certainly incomplete, but still it indicates how broad the spectrum of capabilities of a powerful mathematical assistant system needs to be. Due to this wide scope of requirements and the resulting multitude of interfaces between different aspects that all need to be integrated in a coherent manner, most of the currently available systems focus on one or just a few of the aspects listed above. A true mathematical assistant system should at least in principal have the capacity to cover all of the tasks above in a systematic and meaningful manner[1].

[1] There are, of course, specialized programs for special areas of mathematics, that are heavily used by mathematicians engaged in these areas. Examples of this species are Matlab for numerical computations and simulation or the *R* package for statistics. We do not want to consider these as mathematical assistant systems in this context. Neither do we want to consider programs like LaTeX or spreadsheet programs although they might assist the mathematician in her everyday work.

One category of general-purpose systems serving as sort of mathematical assistants in the sense described above are *computer algebra systems* with Mathematica, Maple, Derive, Maxima, or Sage as the most popular representatives. These systems are typically based upon a computer-representation of certain predefined mathematical objects (e.g. polynomials, matrices, special functions, etc.) together with a large collection of algorithms to perform computations involving these objects and a programming environment allowing for user-defined system extensions. These systems can be employed for proving statements only in the case, where the statement in question can be transformed into an equivalent computational problem that can be solved using the available algorithms. As an example, many theorems from geometry can be translated into checking the solubility of a system of polynomial equations, which can be decided algorithmically by testing whether the reduced Gröbner basis of the polynomials is $\{1\}$. These systems, however, almost completely lack any possibility of logical reasoning in the style how a mathematician would present a proof using "pencil and paper".

Over the past sixty years[2], *automated theorem proving systems* have been developed. These systems are based on logic and they perform logical steps in order to synthesize a proof. Roughly, these systems can be divided into *fully automated provers* and *interactive provers*, sometimes also called *proof assistants*[3]. In essence, automated theorem provers consist of a collection of logical rules and an engine that applies these rules in a way such that the repeated rule application constitutes a proof of the statement in question. Popular fully automated provers are e.g. Vampire, Spass, or E, the most widely used interactive provers seem to be Isabelle, Mizar, HOL/light, and Coq. Most theorem provers lack powerful computation mechanisms, which limits their applicability as mathematical assistant systems in the above sense.

The *Theorema system*[4] tries to bridge this gap between computer algebra systems and theorem provers by providing both reasoning and computation within the same language. From the very beginning it has always been a major design focus in the Theorema system to support the working mathematician during all facets of her work. One crucial aspect of this support is to combine reasoning, computation, and the formulation of algorithms and their execution within one logical frame. These capabilities form the core of a mathematical assistant system, and together with natural style input and output they are the core features, in which Theorema stands out from the majority of systems available.

Theory exploration is a direction of research that has emerged over the past

[2] 2014 marks the sixtieth anniversary of the first automated proof, which proved that the sum of two even numbers is even. It was generated by a computer program based on a decision algorithm for a fragment of natural number arithmetic invented by Mojżesz Presburger and implemented by Martin Davis in 1954.

[3] For this similarity in naming, mathematical assistant systems are often wrongly believed to be mere proving systems.

[4] The Theorema project is the materialization of Bruno Buchberger's view on mathematics and computer science. In its present form as a system implemented on the basis of Mathematica, it was conceived by Buchberger in 1994, when he designed the fundamental system components and sketched the new system in a prototype implementation based on an α-test version of the then new Mathematica 3, thus, 2014 marking the twentieth anniversary of Theorema. This prototype already included the executable Theorema language, a basic predicate logic reasoner, and an induction prover for natural numbers. He then initiated regular seminar meetings under the title "New Math", which gradually turned into the now regular Theorema seminars.

ten to fifteen years[5]. The Theorema-approach to theory exploration is as follows: as soon as a new concept (i.e. a new object or a new property) is introduced by definition, we investigate and prove all interactions of the new concept with already available ones. These lemmas are not always of deep interest as such, but they constitute a saturated knowledge base for proofs of more difficult and more interesting statements when the theory evolves. Therefore, the automation of these proofs is crucial in order not to distract from the really important theorems. Moreover, the automation of complex proofs becomes more tractable, when a saturated knowledge base is at our disposal. The philosophy of theory exploration is inspired by how mathematics is done "in real life", in sharp contrast to the tradition in automated theorem proving, which over the years strived to prove isolated theorems from the theory's axioms.

Advocating a style of exploration makes Theorema particularly well suited for use in *math education*, because it puts emphasis on not only that theorems are indeed correct but also on where they come from and why they are formulated in the way they are. The main reason, however, why Theorema is a powerful tool to support math teaching, is the system's natural-style interface. Theorema understands, on the one hand, mathematical input expressed using a variety of two-dimensional notations established in textbook mathematics. When used in a finitary context, such expressions can immediately be used in computations. On the other hand, Theorema generates all output in a style how a well-educated mathematician would write it. This holds in particular for (most of the) automatically generated proofs by the system, which can be displayed showing the steps of the proof in a customizable level of detail with explanatory text in a language of choice. With this facility the Theorema system can be employed for instance as a proof trainer for beginning students of mathematics or future mathematics teachers.

The author has contributed significant shares to the Theorema system throughout its entire development, in particular in

- basic software engineering concerning the system layout for large Mathematica software components developed in a distributed team,
- designing and implementing the computational layer of the Theorema language,
- planning and realizing the organizational and the user layer of the Theorema language,
- developing general and special proving techniques within the Theorema system,
- integrating of computation into reasoning components,
- the conception and programming of novel user interface components,
- the application of the Theorema system in the formalization and exploration of mathematical theories, and

[5]The term "theory exploration" was coined by Bruno Buchberger in the frame of the European Calculemus Network project in the keynote talk at the Calculemus Conference 1999 in Trento, Italy. It marks a shift of paradigms away from proving isolated theorems, which was the predominant approach in the automated theorem proving community, towards the conception of "mathematical assistant systems" rather than just automated theorem provers.

– the application of the Theorema system in math education at university, college, and high school level.

This thesis summarizes the author's publications referring to some of these topics. The papers appeared in refereed journals or refereed conference proceedings of established conferences and workshops in the areas of symbolic computation, automated reasoning, mathematical knowledge management, or mathematics education. Section 2 focuses on design and implementation of mathematical assistant systems, in Section 3 we discuss the use of Theorema in theory exploration and mathematics education, and finally Section 4 investigates the mutual influence between mathematical assistant systems and mathematics education.

The accumulated experience in design, implementation, and application of a mathematical assistant system is reflected in the new implementation of the Theorema system, called Theorema 2.0, which has been undertaken essentially by the author of this thesis[6]. Theorema 2.0 is available under GPL license.

2 Design and Implementation of Mathematical Assistant Systems

In this section, we collect articles concerning the *design, implementation, and application of mathematical assistant systems*. In this area, the contributions range from *basic system design* over design and implementation of *general and specialized inference methods*, the *realization of user interface components* for mathematical assistants, and case studies in the *formalization of mathematics* based on a mathematical assistant system.

One aspect, which plays a crucial role in several of the authors contributions in this area, concerns the integration of reasoning and computation. In this context, *computation* means the execution of algorithms performing operations on mathematical objects that can be represented in an extensional form through data structures. The Theorema language provides data structures for some basic mathematical entities built into the system, such as

– numbers (basically all kinds of numbers available in the Mathematica programming language),

– tuples and finite sets (both represented through list-based data structure in the Mathematica programming language), and

– predicate logic expressions (represented through general expressions in the Mathematica programming language).

Based on these structures, a fragment of the Theorema language possesses computational semantics given by algorithms performing operations on these objects. In the case of numbers, these are the algorithms for the arithmetic operations, whereas for tuples and finite sets these are operations such as tuple concatenation, finite set intersection, or finite set construction using set builder

[6] B. Buchberger, W. Windsteiger, T. Jebelean, and T. Kutsia. Theorema 2.0. Source code publically available under GPL license, `http://risc.jku.at/research/theorema/software/`, February 2014.

notations such as $\{f[i] \mid_{i=1,\ldots,10} P[i]\}$, and for logical formulas these range from simple algorithms for boolean connectives to algorithms for processing quantifier expressions with finite ranges such as $\forall_{i=1,\ldots,10} P[i]$.

In the Theorema system these algorithms are accessible for the user on the top-level, an aspect in which Theorema tries to mimic typical computer algebra systems. Moreover, computation can also handle the unfolding of definitions given by the user allowing for patterns involving sequence variables and recursion. A challenge for the system design, which will be addressed in several contributions below, is then

- to integrate computation facilities into the reasoning engines in a coherent way such that computations occurring inside a proof are done in the same way as on the top level and
- to provide the user full control over which built-in algorithms may be accessed during a computation or a proof.

Paper #1: Journal of Symbolic Computation 41 (3–4):435–470, 2006. This paper introduces the theoretical foundations and the implementation of the Theorema set theory prover including examples of fully automated proofs generated by it. Following the prover design of Theorema 1 the prover consists of several so-called *special provers* that are combined into one prove method, which is available for the user, the so called *set theory user prover*. The provers support reasoning involving set-theoretic concepts introduced through the Zermelo-Fraenkel axioms of set theory (ZF) and definitions based thereon. On the language level, the Theorema system has always supported a syntax for sets inspired by the common sense use of set theory in mathematics. In addition there is computational semantics for finite sets, which are represented by a list-based data structure in the Mathematica programming language. Semantics for finite sets is given through algorithms performing finite set operations such as set construction, membership, union, intersection, power set, and the like.

With this prover, we added semantics for infinite sets in the Theorema system on the reasoning level. A consistent fragment of language, whose semantics can be given by inference rules based on the axioms of ZF, has been clearly identified within the naive set theory language of Theorema. The respective inference rules have all been implemented and integrated into the proving machinery of the Theorema system. The novelty in this prover compared to other systems is that in addition to standard inference rules reflecting the axioms of ZF, the prover also contains specialized inference rules for special cases, which condense frequently occurring proof patterns to just one step. As a simple example, instead of reducing '$\emptyset \subseteq X$' by definition to '$\forall x : x \in \emptyset \Rightarrow x \in X$' and then let predicate logic reasoning complete the job in a couple of steps, the Theorema set theory prover simplifies '$\emptyset \subseteq X$' to 'true' in just one step. Following this philosophy, the prover generates proofs in a style like a well-trained mathematician would give the proof.

From the point of view of prover design, the novelty in this prover is a general mechanism to seamlessly integrate the simplification of formulas and terms based on computation as described above. The mechanism is *general* in the sense that it does not only apply to computations with finite sets but to all kinds of computations supported in the Theorema language. Although the original

desire when implementing the set theory prover was to integrate computations with finite sets into Theorema's reasoning mechanism, it turned out that all sorts of computations available in the Theorema language (numbers, tuples, quantifiers with finite ranges, etc.) can be made accessible for reasoning in the same fashion. This makes this prover also a suitable tool for automating proofs for simple theorems in high school or undergraduate math, because these often involve arithmetic on numbers. In this context, powerful resolution provers often fail to prove simple high school theorems due to their inability to perform simple computations with numbers.

Yet a user need not rely on computation when it comes to finite sets. The prover also provides reasoning capabilities for finite sets, which are derived from the same foundations as the finite set computations mentioned above, since both rely on ZF. It is a matter of prover configuration when the user calls the prover, whether the handling of finite sets is based on finite set computation or finite set reasoning. Reasoning typically shows more detail, whereas computation is faster and generates fewer steps. It's up to the user what she prefers.

It should be noted that computation can handle certain cases even involving infinite sets. Due to the representation of integer numbers in a data structure in the Mathematica language, questions like $7 \in \mathbb{N}$ can of course be easily decided. Needless to mention that the computation power with respect to infinite sets is rather limited.

In the frame of this work, extensive case studies have been carried out using the set theory prover. The capabilities of the prover are demonstrated by example proofs taken from these formalizations.

Paper #2: Journal of Applied Logic 4 (4):470–504, 2006. The work is a joint publication of the Theorema group describing the project progress accomplished between the years 2001 and 2005. The main contribution to this publication lies in the conception and implementation of a new specialized reasoning method, called the *BasicReasoner* in the Theorema system. The main aim of this method is to generate easy-to-read proofs for standard high school or undergraduate proofs, which are many times based on a rather simple logical structure combined with arithmetic on numbers and simplification of arithmetic terms. Due to the fact that many automated reasoning systems cover only the logic aspects of proving but totally lack algorithms for computation with numbers (see also the discussion of Paper 1 above), there is a considerable "market" for such provers, in particular if they should be applied in math education.

The BasicReasoner makes use of the general mechanism for interfacing reasoners with computation described above. In addition to pure execution of arithmetic operations as used in Paper 1, we developed an interface to more powerful symbolic computation algorithms for arithmetic expressions available in the underlying Mathematica system. In order for this to work reliably, we employ a translation between Theorema expressions and their corresponding Mathematica representations, such that, upon user request, arithmetic expressions can be passed through symbolic manipulation functions such as *Simplify* or *FullSimplify*. It should be noted, that Theorema operations are strictly separated from Mathematica operations, such that any application of Mathematica's black-box-algorithms can only happen on explicit user request.

The BasicReasoner has then been used in a formalization of the proof of the irrationality of $\sqrt{2}$, which was a benchmark problem for mathematical assistant systems at that time, see also Paper 3. Moreover, many of the author's contributions to the Theorema system during the years covered in this report were of foundational nature and helped the system to mature. Some of them are mentioned only briefly in the introduction, like the adaption of a functor mechanism for computation, others are used in other sections without focusing on them, like the design of the user language referred to in theory exploration.

Paper #3: *LNAI* Volume 3600, pp. 96–107, 2006. This paper is the Theorema solution to a benchmark challenge for automated provers and mathematical assistant systems posed by Freek Wiedijk. The task was to formalize a proof of the irrationality of $\sqrt{2}$ using a method that is most suitable for the system in use thereby advocating how the system is intended to be used by the developers. There were no requirements on whether the formalization is done in first order or higher order logic, whether one wants to use set theory or not, nor which additional knowledge is applied in the proof. The only requirement was that all lemmas used to make the proof go through must also be proved in the system. This was a perfect setting for theory exploration in the area of integers, rational numbers, and divisibility.

The formalization presented in this paper shows all details as requested including the proofs almost exactly in the form as generated by the system. In fact, the generated proofs had to be translated into LaTeX due to formatting requirements from the publisher. Because of this, part of the power of the proof presentation in Theorema has been lost, because originally the proofs are rendered as Mathematica notebooks with math formatting equally powerful as LaTeX, but using hyperlinked labels and colors in addition.

We have elaborated various approaches to proving the irrationality of $\sqrt{2}$ differing mainly in the definitions of "irrationality" and "sqrt". One approach was to define $\mathbb{Q} := \{\frac{a}{b} \mid a, b \in \mathbb{Z} \wedge b > 0 \wedge \text{coprime}[a, b]\}$, i.e. defining the set of rational numbers through canonical representatives with co-prime numerator and denominator, and then stating the theorem simply as $\sqrt{2} \notin \mathbb{Q}$. This approach requires set theory, which would be a nice fit for the Theorema set theory prover as presented in Paper 1. Although this formulation, in particular the statement of the theorem, is probably the most natural one and the one most appreciated by many mathematicians, we decided not to pursue this path further because we finally considered the use of set theory as rather "artificial" in this case because immediately after substituting the definition of $\mathbb{Q}$ there is only one step involving set theory, namely expanding membership in a set specified by a set builder. From then on, no set theory is entering the stage anymore, and the set theory step is by far not the most interesting part in this proof. Instead, we defined a predicate "rat" expressing the rationality of a positive real number using a similar property like it has been used in the definition of $\mathbb{Q}$, and then stated the theorem as $\neg\text{rat}[\sqrt{2}]$. Using this approach shortens the proof by three steps, on the other hand it showed that regardless how we setup the stage, the Theorema system can generate the necessary (uninteresting) steps without problems.

The novelty in this prover is that we added reasoning capabilities for the Hilbert ε-quantifier. It is the language constructs that describes *such an* x sat-

isfying a given property P, it is a quantifier that binds the variable x. The Theorema language supported the ε-quantifier since the early days but it was used only in computations up to then. The *ElementaryReasoner* used in this formalization was the first reasoner that had proof support for the ε-quantifier built-in. The Theorema language supports two versions of this construct, one written as $\ni$ and the other one written as $\ni!$. The first one stands for the ε-quantifier as described above, the $\ni!$ is a variant asserting uniqueness of the x, it describes *the* uniquely existing x satisfying a given property P. When these formulations are used in Theorema, the system does not require a proof of existence and uniqueness. This follows the general philosophy of the system that not everything written in the system must be proven, the system still allows unproven statements to be used in proofs of higher-level propositions. This is a convenient setting, which allows top-down development of mathematical theories: starting from interesting statements we may develop automated proofs based on some unproven Lemmas; we may then proceed developing the theory "downwards" by proving these Lemmas (again generating unproven Lemmas during their proofs). This strategy should be combined with bottom-up theory exploration, where we start from the axioms of a theory and develop the theory "upwards" by systematically generating and proving properties, how newly introduced notions interact with already existing ones.

Using the $\ni!$-quantifier allows a very convenient definition of $\sqrt{x}$, and the resulting proof again relies on symbolic simplification techniques available in the *ElementaryReasoner* in a fashion already described in Paper 2. The proof relies on one auxiliary lemma, which is proven using the same reasoner. It again uses symbolic simplification techniques involving arithmetic on numbers. Finally, this paper contains a rather detailed description of the philosophy, the logic, and concrete reasoning methods used in the Theorema system.

Paper #4: Proceedings of the Calculemus Conference, pp. 35–50, 2006. This paper originates from the author's work on the Analytica system designed by E. Clarke at Carnegie Mellon University, Pittsburgh, USA. The Analytica system is implemented and based on Mathematica, it was developed in the early 1990s, and it was the first automated theorem prover based on a computer algebra system. The systems was intended to generate proofs in elementary analysis and number theory, where the powerful symbolic computation techniques available in Mathematica can be heavily applied. In the concrete case, we report on the new implementation of the system called Analytica *V* and based on it the formalization of the Mordell-Weil Theorem on the group of rational points on an elliptic curve and a theorem of Dirichlet on group characters.

We contributed to this work the conception and the implementation of a powerful lookup mechanism for efficiently checking the validity of conditions w.r.t. the global knowledge base of Analytica. The mechanism is based on backward reasoning (often called backchaining) based on universally quantified implications. We explain the key idea of this method in its simplest case, where we use an implication $\forall x : (P[x] \Rightarrow Q[x])$ in order to reduce the proof of an instance $Q[a]$ of the right side to proving the respective instance $P[a]$ of the left side. The reduction mechanism is realized through Mathematica programs, which are generated automatically from facts and quantified implications in the knowledge base. An atomic fact of the form $F[a]$ is translated

into a program `F[a]:=True`. As for the implications, the quantified variables are replaced by patterns in order to allow Mathematica's pattern matching capabilities to take care about instantiation. Instead of reducing an instance $Q[a]$ to $P[a]$ by `Q[x_]:=P[x]` we generate the program `Q[x_]/;P[x]:=True`. In this way, we employ Mathematica's backtracking facilities to automatically test all possible reductions of $Q[a]$ by any known implication. Evaluating $Q[a]$ in the presence of these programs in Mathematica invokes a recursive computation that eventually stops and returns `True` in case a fact $F[a]$ is reached, otherwise $Q[a]$ stays unevaluated. Of course, we handle also more complicated implications including conjunctions occurring both on the left and on the right hand side. Equivalences $\forall x : (P[x] \Leftrightarrow Q[x])$ need to be oriented to indicate in which direction they should be used for reduction, since using both directions would lead to infinite recursion. We employ this mechanism for the case explicit predicate definitions of the form $\forall x : (P[x] :\Leftrightarrow Q[x])$, since here the decision for an orientation $\forall x : (P[x] \Rightarrow Q[x])$ is straight-forward. This is yet another example for the interaction of proving and computation.

Analytica *V* also features a simplification procedure for groups, which computes canonical forms for group terms based on associativity, commutativity, identity, and inverse elements. This module plays a crucial role in the formalization and the automated proof of the Dirichlet theorem on group characters. Finally, in the course of formalizing the Weak Mordell-Weil Theorem, we developed a decision method for detecting contradictory assumptions based on the Gröbner bases method. For proving associativity of the group law for rational points on an elliptic curve, a huge case distinction needs to be setup. In each of the cases, the preconditions are described by polynomial equalities, and the Gröbner bases method is employed to immediately eliminate cases containing contradicting conditions.

Although Analytica *V* does not maintain a proof object, we developed a method for visualizing a proof generated automatically in the system by recording the individual steps that the prover performed. From the recorded information, a proof presentation in natural language can be generated similar to how proofs are presented in the Theorema system.

Paper #5: Electronic Proceedings in Theoretical Computer Science Volume 118, pp. 72–82, 2012. This paper marks the first presentation of Theorema 2.0, the newly designed and implemented version of the Theorema system[7]. Since the Theorema system began to mature during the first half of the last decade, we started to give away the system to other mathematicians outside the Theorema developers' team and employ the system in cooperation projects outside the core Theorema group. On the one hand, we cooperated in the formalization of mathematical theories brought to our attention by colleagues working in these areas, see some of the papers in Section 3. On the other hand, we engaged in projects concerning the teaching of mathematics, in particular on computer-supported and logic-emphasizing teaching of maths, see some of the papers in Section 4.

In these cooperations it became more and more evident, that for the acceptance of a mathematical assistant an easy to use interface is of utmost im-

[7] This description will be presented in a bit more detail, because due to space limitations only an incomplete account of the new developments in Theorema 2.0 could be given in the paper.

portance. While an attractive interface is a vital component in any computer system and any product in a more general sense, we feel that this aspect gains additional weight in the context of mathematical assistant systems due to the intrinsic difficulty in the task of calling an automated theorem prover: the proof goal has to be spelled out in a logically rigid manner, the knowledge to be used in the proof needs to be formulated and specified, and the proof method needs to be determined. The more versatile a prover is, the more possibilities for adjusting its behaviour it typically offers and the more sophisticated the handling then usually becomes.

The maintenance of the original implementation based on concepts dating back to the mid-nineties of the past century became increasingly troublesome, and the flexibility of the system with respect to the interplay of different system components turned out to be insufficient. We therefore decided to base the system on a completely new implementation, whose main aim was to preserve the numerous powerful facets, which proved successful in the original system, enhance several components, which performed in an unsatisfying way, and accompany all with an up-to-date graphical user interface (GUI). We have already gained experience in developing GUI components for Theorema 1 earlier, see the Papers 10, 11, and 12 in Section 4, where we used Wolfram Research's *GUI-Kit for Mathematica* for the development of rather powerful application widgets. The GUI-Kit provided an interface that allowed calling certain Java functions from within Mathematica. The drawbacks of this approach range from a rather cumbersome implementation of GUI components, through platform-specific behaviour depending on the installation of the underlying Java engine, to poor performance in terms of reactivity of interactive widgets due to the communication between Mathematica and Java.

With the release of Mathematica 6 in 2007 and Mathematica 7 in 2008 and the integration of dynamic objects into the standard Mathematica programming language, an implementation platform for a Theorema-GUI was accessible within Mathematica. The chief advantages over the GUI-Kit are the ease of implementation due to the seamless integration of GUI elements into the Mathematica language and a faster, more reliable, and system independent performance on all platforms supported by Mathematica. It was the starting point for a complete re-design and re-implementation of the Theorema system. The basic data structures have been re-modeled, the generic proof search algorithm has been re-organized, and the implementation model for logical inference rules has been setup in a new form. Due to these fundamental changes, no single piece of code from Theorema 1 could move into the new implementation.

The user-system-interaction model changed completely between Theorema 1 and Theorema 2.0. The original Theorema 1 followed the command-line function call paradigm known from Mathematica, meaning that every action is a Mathematica command written into a notebook and executed by pressing Shift-Enter. In this spirit, stating a mathematical definition in a Theorema session means executing a command `Definition[...]`, and with a smart design of the Theorema language we took care that when reading the whole command and thereby ignoring all brackets '[' and ']' also within the '...' (including informal structuring elements such as labels or theorem/definition names), it reads almost like a definition in a math textbook. In the same vein, initiating an automated proof can be accomplished by evaluating a command `Prove[...]`, just that this is a mathematical assistant typical expression that has no real counter-

part in the textbook world. In particular the Prove-commands are typically quite involved owing to the multitude of specifications needed to initiate a successful automated proof.

Theorema 2.0 propagates a point-and-click interaction scheme as known from navigating through web shops or working with a state-of-the-art digital image processing program. What needs to be written in a formal way into a notebook are just the plain formulas making up the core of a definition or a theorem, all informal parts are not anymore part of the formal text. Calling a prover is governed through a fully mouse-driven "wizard" similar to what people find familiar from typical software installation procedures. This wizard is part of the new main Theorema GUI called the *Theorema commander*, alongside with components supporting the two-dimensional input of mathematical expressions, organization of the Theorema session, administration of knowledge archives, the preparation of computations, adjusting system preferences, and the like. It guides the user step-by-step through the prover configuration process,

- from composing the knowledge base, through
- setting up built-in knowledge available in computations during the proof (see the introduction to Section 2 for an outline on the interplay between proving and computation in Theorema),
- synthesizing the prover from the available inference rule collections, and
- selecting the proof strategy, to finally
- fine-tune the desired prover output, and
- setting global parameters such as the maximum search depth or the maximum time provided to complete the proof.

The generated proof, even if not successful, is presented in two forms simultaneously, on the one hand in a Mathematica notebook as structured text with language dependent explanations in textbook-style, and on the other hand as a graphical visualization of the proof tree, which corresponds to the proof generation process. The two representations are bi-directionally linked such that clicking at a location in one representation will indicate the corresponding part in the other location. These features prove helpful in particular when failing proofs are inspected in order to figure out the reasons for failure.

The components of the Theorema commander make heavy use of Mathematica's dynamic objects that allow the implementation of modern graphical user interface components like sliders, menues, checkboxes, tooltips, or radio buttons inside the Mathematica language without calling external Java routines. In expectation of the system's use in math education, the interface also provides a virtual keyboard that should be of use when working with Theorema on a digital white-board or a tablet device. In addition, Theorema 2 caters for interactive proving in the heart of its proof search engine.

Much work has been invested also in the organization of Theorema documents that carry the formal parts. This aspect was mainly driven by the application of Theorema 2 described in Paper 9. Worth mentioning is the introduction of the novel concepts of global declarations and global conditions. The first

one serves the task of declaring a symbol as a universally quantified variable in a certain part of the document, the second one attaches a condition to every formula occurring in a certain part of the document. They are written as a universal quantifier lacking the body formula or an implication lacking its right hand side, respectively. Their semantics relies on a careful control over the nesting structure of various formal and informal document blocks in order to maintain clearly defined scopes for the global entities.

Theorema users may also turn into Theorema developers and work in both roles at the same time. We expect mathematicians working in specialized areas of mathematics to code their specialized techniques applicable in these areas as a special prover modules in the frame of the Theorema system. We therefore pay heavy attention that the system architecture and the programming style in Theorema 2.0 stays as easy to understand as possible also for non-expert Mathematica programmers. In addition, we make Theorema 2.0 publically available under GPL license.

3 The Use of Mathematical Assistants in Theory Exploration and Education

In this section, we collect articles concerning the *use of a mathematical assistant system* (the Theorema system) in theory exploration and mathematics education at university level.

Paper #6: Proceedings of the Calculemus Conference, pp. 28–47, 2001. This paper reports on the fully automated complete proof of the Mutilated Checkerboard Theorem. The theorem was originally formulated in 1964 by John McCarthy and states that an 8×8-chessboard with two opposite corner fields removed cannot be fully covered with dominoes. The theorem became famous as a benchmark problem for automated theorem provers and the problem was solved independently based on various proving systems. Since the formulation of an informal four-line proof based on set theory by McCarthy himself in 1995, the problem is a standard benchmark problem *for set theory provers*[8] in particular.

The solution to the checkerboard problem given in this paper is based on theory exploration using the Theorema system and its set theory prover described in Paper 1. The problem formulation is straight-forward since the Theorema language covers all set theory language constructs typically in use in mathematics. The McCarthy proof represents the chessboard as $\{1, \ldots, 8\} \times \{1, \ldots, 8\}$ and relies on auxiliary concepts “adjacent”, “domino-on-board”, “partial-covering”, and “color”, which are defined in the language of set theory and used in the proof in an intuitive manner. These concepts are given to Theorema as explicit definitions and, of course, the set theory prover has no built-in knowledge about these newly introduced notions. Hence, we add short exploration rounds after introducing each of the definitions. In a first phase, we use computation with concrete examples in order to clarify whether the given definitions correctly reflect the intended meaning. During the second phase, we prove simple lemmas

[8] McCarthy: “While no present system that I know of will accept either the formal description or the proof, I claim that both should be admitted in any heavy duty set theory.”

expressing intuitive properties of the new entities. In this approach it proves very useful that computation and proving can be done in the same language frame on the user level; the same formulation that is used in the proofs can also be used for computation on finite input without any conversion whatsoever.

The definitions for adjacency and color proposed in the McCarthy-proof use arithmetic on integers based on the coordinates of a field on the board. Well-known functions such as difference, absolute value, and integer remainder are employed. Together with the chessboard as a finite set this forms the perfect playground for the application of built-in integer arithmetic and finite set arithmetic in some parts of the proofs. In particular, using the knowledge proved in earlier exploration rounds, the proof of the final theorem boils down to a proof by contradiction assuming that the cardinality of the set of all white fields equals the cardinality of the set of all black fields on the mutilated board. Given the concrete 8×8-size of the board, both the sets above and the inequality of their cardinalities can be obtained by Theorema's computation facilities in the proof.

The novelty in this paper is that it reports on the formalization of a piece of mathematics using the theory exploration paradigm in the Theorema system. It solves a proof problem that is widely acknowledged as a benchmark problem for mathematical assistant systems. Moreover, in addition to core set theory reasoning the paper demonstrates Theorema's capabilities with respect to the integration of proving and computing on both the user-level as well as on the reasoning level.

Parts of this formalization have been used over the time in the university course "Predicate Logic as a Working Language" for first-year students of mathematics at JKU Linz. In this course, we teach the rigorous use of predicate logic and basic set theory as the fundamental basis for mathematics and advocate a style of developing bigger theories step by step from smaller building blocks. The key insight that we want the students to learn with this approach is that single proofs become structurally easier when being built upon carefully structured knowledge that has already been proved earlier. The ideas of theory exploration in the Theorema system thus strongly influence the design of math education in this area. As for proving, we teach students to apply logical proof rules that are based on the syntactical structure of the proof goal and the knowledge base. The Theorema provers follow the same philosophy, which allowed us to present automatically generated proofs as sample solutions to proof problems given in the lecture. Occasionally we tried to get students to work with Theorema 1 themselves, but it turned out that the system was too difficult for them to use at that stage, which fired our desire to develop an improved interface that finally led to the design and realization of Theorema 2.

Paper #7: Proceedings of the Calculemus Conference, pp. 130–136, 2003. This paper presents the formalization of an algorithm for univariate polynomial interpolation. The domain of univariate polynomials over a coefficient field K is introduced through a functor in the Theorema system. In Theorema, the functor construct allows to construct new domains from given ones by defining operations in the new domain based on already existing operations in the given domains[9]. In the case of univariate polynomials, polynomials are represented as

[9]The word "functor" is used like in the ML programming language rather than in category theory.

coefficient tuples and the polynomial operations are defined in terms of existing operations on the coefficients. Theorema functors are inherently algorithmic, the newly introduced operations can immediately be used in the computational framework of Theorema to do computations in the new domain.

In the concrete example, the polynomial functor defines the abstract domain `Poly[K]` of univariate polynomials over some coefficient domain `K` including basic polynomial arithmetic and polynomial evaluation. The polynomial operations are defined using the language of predicate logic with standard textbook-like notation. They need not be written using a concrete programming language, but still the operations can immediately be executed in Theorema computations. On the other hand, due to their representation in standard predicate logic, they can directly be used in reasoning. In the paper, this is demonstrated with a recursive algorithm for polynomial interpolation, the so-called *Neville algorithm*.

The novel part of this paper deals with the correctness proof of the algorithm. The assertion is a statement about *all tuples* x and a, we therefore employ the *tuple induction prover* available in the Theorema system. In order for this case study to succeed, the prover had to be enhanced in several respects:

1. The induction base is normally the empty tuple. We implemented an enhanced choice of the induction base that uses additional information about the actual range available in the knowledge base for a suitable induction base. In the Neville algorithm, we know $|x| > 0$, where $|x|$ is Theorema syntax for the "length of the tuple x", hence the prover chooses to prove the statement for all tuples of length 1 as the base case.

2. The prover is enriched by inference rules encoding knowledge about tuple length. As an example, the prover can infer form $|a| = 1$ a concrete representation of a as $\langle a1 \rangle$, which is then used to replace a in the proof goal such that the base case of the Neville algorithm applies. The resulting statement can easily be proven by just executing polynomial operations defined in the functor.

3. In the step case, the knowledge $|x| > 1$ and $|a| = |x|$ is used to represent x as $\langle x0, \bar{x}, xn \rangle$ and a as $\langle a0, \bar{a}, an \rangle$, respectively. This representation is not hard-coded in the prover but it is *chosen* by the prover after inspecting the recursive definition of the algorithm to fit the representation to the recursion scheme used in the algorithm. Since the recursive definition of the algorithm can then be applied, the rest of the verification boils down to simple arithmetic properties in the underlying coefficient field K.

The now *verified algorithm* can then be used immediately to compute a concrete interpolation polynomial over $\mathbb{Q}$.

We claim this as a perfect environment for *teaching mathematical algorithms*, since definition of domains, definition of algorithms, reasoning about algorithms, and their execution can all be done within one language and one logical frame. The presentation of algorithmic mathematics is then not distracted by program code referring to a particular programming language including all syntactic details. In contrast to giving algorithms in pseudo-code, the algorithms in the Theorema language can immediately be used to compute concrete examples.

This approach of teaching algorithms was pursued when inventing the course "Algorithmic Methods" for first-semester students of mathematics at JKU Linz. Again, the capabilities of the mathematical assistant system have a strong influence on the teaching of mathematics in this area. On the other hand, the use of the system in teaching also influences the development of the system towards better usability by non-experts, and it gives rise to interesting case studies like the correctness proof by tuple induction presented in this paper, which was originally neither part of the course nor presented in class. We will present more examples on the mutual influence between mathematical assistant system development and education in Section 4.

Paper #8: *LNAI* Volume 6824, pp. 58–73, 2011. In this paper, we present the formalization of a fragment of theoretical economics using the Theorema system. Mathematical economics has been identified as an application area for formalized mathematics and its computer support. As a first step in this direction, we investigated so-called *pillage games*, which are used in economics to describe power relations between opponents. The formalization is based upon a paper by the two co-authors and it comprises fully automated proofs of three lemmas from that paper and the formulation of an algorithm for computing the stable set in the executable language of Theorema.

The novel contribution in this paper is the use of the Theorema system in the formalization of a new area for formal mathematics. Up to the authors' knowledge, no mathematical assistant systems have ever been applied in theoretical economics in order to formally verify the theorems and algorithms on which decisions in one of today's most important application areas of mathematics are based. This work constitutes a first step into this new field, thereby also demonstrating to economists the feasibility of available methods and systems to be applied as a future standard tool in their everyday work. Mathematical assistant systems have not yet reached the level of acceptance like computer algebra systems. It needs many examples where their successful use can be demonstrated, both to mathematicians and to users of mathematics in application areas.

The second novelty in this work is the use of the computational facilities of the Theorema system. The main result in the paper, on which this formalization is based, is a pseudo-algorithm for determining the stable set in a pillage game with three agents and powerfunctions satisfying certain properties. Parts of this procedure are based solely on finite sets and integer arithmetic and are thus algorithmic, whereas other parts are non-computational due to the infiniteness of certain sets involved when testing conditions in the procedure. Since Theorema allows to specify the knowledge base to be used during a computation in exactly the same way as we may specify the knowledge available in a proof, we can provide the computation with auxiliary lemmas that allow the decisions to be made in an algorithmic fashion. For instance, if the non-algorithmic condition $c[a]$ in the procedure needs to be evaluated, Theorema offers various possibilities:

1. Provide a lemma stating $c[a]$ or $\forall x : c[x]$, which will both allow the condition $c[a]$ to evaluate to `True` and thereby continue the computation.

2. Provide a lemma stating $c[a] \Leftrightarrow p[a]$ or $\forall x : (c[x] \Leftrightarrow p[x])$, which will allow the condition $c[a]$ to evaluate to $p[a]$ and thereby continue the computation in case $p[a]$ is algorithmic.

In the examples at hand, the stable set could actually be computed by this "algorithm" for three well-known powerfunctions used in practice. This is yet another instance of the unique integration of reasoning and computation offered in the Theorema system.

Auxiliary lemmas can be proved before or after they are used. However, there is no obligation for them to be verified, which gives a pragmatic approach to theory exploration, since the user can decide which knowledge she trusts. Of course, everything that is explored later is relative to the unproved lemmas thrown into the theory. In practice, this is very convenient because it allows to concentrate on the interesting theorems before and prove them based on certain lemmas. We can then continue to prove the lemmas and "explore the theory backwards" starting from interesting theorems. Ideally, this complements the "forward exploration starting from axioms and definitions" to arrive at a saturated theory when the two branches meet. In this formalization, backward exploration was heavily used. The possibility to inspect incomplete or failing proofs in the Theorema system turned out to be a key feature for this approach to become practically useful.

Paper #9: *LNAI* Volume 7961, pp. 200–215, 2013. This work is in some sense the continuation of the collaboration initiated in Paper 8. The topic of formalization is again mathematical economics, but this time we deal with auctions, a hot topic in contemporary economics. Formalizing the famous Vickrey theorem from 1961 about second-price auctions is intended to serve as the basis of a future toolbox for auction designers inclined to employ mathematical assistant systems when they design new auction schemes. In the course of deciding for a platform on which to base this toolbox, several systems on the market have been compared with respect to several criteria that might be decisive for the future acceptance of mathematical assistants by auction theorists.

This formalization marks the first real-world application of the then new Theorema 2 system, which was just in statu nascendi when the comparison was initiated. For this reason the proofs have not yet been completed at the time of this publication and the comparison of Theorema is limited to the part of writing the formalization and reading the resulting proofs, where the latter is judged on the basis of reading and inspecting sample proofs that could already be generated with the limited proving capabilities implemented at that time.

The design of the Theorema 2 GUI was heavily influenced by the use of the system in this case study. Most of the new interface features have already been described in more detail in Paper 5, notably the global quantifiers and global implication and their proper processing. Meanwhile there is also a possibility of filtering and searching in the knowledge base browser and the inference rule browser, which were both conceived while elaborating the auction theory formalization.

Similar to the advances described in Paper 8, the novel contribution in this paper is the use of Theorema 2 in yet another formalization of a piece of mathematics. The popularization of mathematical assistants can only progress, if their successful use is demonstrated in as many and as diverse areas of mathematics and its applications.

4 The Interplay between Mathematical Assistant Systems and Mathematics Education

In this section, we collect articles dealing with the *design and implementation of specialized components for mathematical assistant systems* that are specially tailored for the *application in mathematics education*. In particular, we address the mutual influence between mathematical assistants and mathematics education.

Paper #10: Proceedings of the ICTMT8 Conference, 5 pages, 2007. In this paper we report on the development of a learning and teaching environment for mathematics that aims at fostering mathematical creativity on the students' side on the one hand and on the other hand emphasizes the necessity of logically strict argumentation in everyday mathematics. The underlying credo is that carefully designed computer-support can help to reach both goals. The approach taken is to combine two available software packages both implemented based on the Mathematica computer algebra system, namely MeetMath and Theorema. The didactic principle followed in MeetMath is an approach of self-paced learning that tries to provide an understanding of mathematical concepts through computational and graphical interaction. Formal rigidity in the formulation of mathematical statements and support in proving mathematical propositions is the contribution of the Theorema system.

The novelty in this work is the integration of the Theorema system into the didactic framework promoted by the MeetMath system. The task of Theorema in the learning units is at least two-fold:

1. to serve the phase of *strengthening*, where the students should intensify their understanding of mathematical concepts, which they have acquired during computer supported experiments, by being supplied with a logical proof of the correctness of their conjectures;

2. to serve the phase of *acquisition* of new mathematical skills, namely working in a formal language and a rigid system, by being forced to formally spelling out their conjectures in a mathematical language and rigorously prove their conjectures with the help of Theorema.

After having specified the didactic framework, the realization of the above goals required the design and implementation of new interface components for the Theorema system. As already mentioned in Paper 5, calling a prover in Theorema 1 was a quite tricky task that required some expert knowledge on the user's side. We therefore tried to avoid to confront the students with any direct prover call, which we achieved by inventing the so-called *prove panel*, an interactive user interface component based on Mathematica's GUI-Kit. Using this interface, the student can interactively compose the knowledge base for the prover by mouse-click and then call the prover by pressing a button. The buttons are programmed individually in such a way that the appropriate prover will be called with options set properly for the proof at hand. The knowledge base composer implemented for this purpose can be seen as a forerunner of the knowledge base browser in Theorema 2. Besides the different implementation basis, however, the new tools in Theorema 2 are more general, since interface

components in the learning units were always implemented specifically for concrete examples. More details on these developments are given in Paper 11.

This paper is an instance of the bi-directional influence between mathematical assistants and mathematics education. In one direction the availability of an assistant allowed completely new didactic paths to be explored and new learning tools to be invented. In the opposite direction, the use of the assistant system inside a teaching environment has clearly influenced the system design. More than that it has opened new fields for research on system and interface design, which were all taken into account in the new implementation of Theorema 2.

Paper #11: Symbolic Computation and Education, pp. 94–114, 2007. This paper gives a more detailed account on the didactic perspectives of using a mathematical assistant systems in the teaching of university mathematics. The approach of "experimental formal mathematics" is presented, in which we escape the traditional discussion in didactics when it comes to the use of computers: experimental mathematics based on computer-support *versus* formal mathematics based on proofs. Experimental formal mathematics advocates an approach of alternating phases of experimental flavour with phases of more formal character in math teaching.

Another intended reading of the phrase and thus another didactic path that our approach encourages is to teach formal mathematics in an experimental environment. The Theorema system allows the user to meet the challenge of rigorously proving a mathematical theorem in an experimental setting due to the inherently interactive character of the proof generation workflow in the Theorema system. Although the proof is generated fully automatically once the right prover is called with proper option setting providing an appropriate knowledge base for the prove problem at hand, the process of setting up the prove call is highly interactive. Proofs may fail and after inspecting a failing proof the prover might be restarted with modified settings until a successful proof is obtained. This interaction pattern has already been observed in Paper 8, although the report there was on an application scenario and not in an educational setting.

We have enriched the Theorema system with interactive elements that further enhance the system's potential towards serving as a tool for experimental formal mathematics. In order not to just automatize the process of proof generation we identified user interactions during the proof generation process that should be emphasized in order to create an interesting learning experience for students. It is widely acknowledged that one of the most dangerous scenarios in using comfortable interactive point-and-click interfaces in education is the habit to thoughtlessly click available buttons until the solution is found. Therefore we tried to design our interactive widgets in such a way that creativity on the side of the students was explicitly challenged. We have put one aspect that always led to difficulties when using automated provers in Theorema into the focus of our attention. It is an aspect that is commonly ignored in pencil-and-paper proofs and is in general totally underestimated when teaching proof techniques, namely the selection of the knowledge base of known propositions allowed to be used in the proof. We developed a dedicated interface for selecting the knowledge before an automated proof is initiated. In the preparation of this interface, the developer of a learning unit has to define a list of propositions available at the time of the proof from which the student can then select the ones to send to

the prover. A more general procedure would be to always present the student all available knowledge. We decided to give the developer the possibility to preselect a sensible subset for the student to select from, because in this way the learning can be better controlled. In contrast, passing the proper options to the prover has been decided to stay completely hidden from the students.

The novel research in this paper concerns the identification of possible didactic models for the fruitful use of mathematical assistant systems in math education. The learning units presented in this paper go far beyond the mere use of an existing system in teaching. A didactic concept and a guideline for designing learning units has been developed and a couple of learning units covering various topics in undergraduate and graduate mathematics have been explored. Concretely, the paper presents the learning unit on equivalence relations in a bit more detail, where the learning unit benefits from reasoning capabilities for set theory available in the Theorema system, see Paper 1. A new aspect is moreover the combination of experimental exploration of mathematics based on interactive visualizations of mathematical concepts with a component of formal mathematics represented by the Theorema system. The implications of the research pursued in this work on the design of the new Theorema 2 are similar to those described in Paper 10.

Paper #12: Proc. Intl. Congress on Math. Education, pp. 351–357, 2008
In this paper, we report on another learning unit developed in the spirit of the system proposed in Papers 10 and 11. The topic this time is univariate polynomial interpolation. In contrast to the approach described in Paper 7, where the algorithm was given and the interest lied in its a-posteriori verification, the emphasis this time is on the derivation of just the decisive properties of polynomial evaluation that pave the way towards an algorithm for solving the interpolation problem. The algorithm derived in this way is standard Lagrangian interpolation. We then discuss different *recursion schemes* that proved useful in the discovery of algorithms in other areas of mathematics. The Neville- and Newton-algorithms for univariate interpolation can be discovered in this way. Consequently, after deriving an interpolation algorithm, we investigate its mathematical properties formally. Unlike in Paper 7, where the correctness of Neville interpolation had been shown by tuple induction, a simplified correctness statement is formulated such that a rather straight-forward simplification proof with case distinction goes through.

In total, we contributed four full learning units to the system, ranging from elementary set theory to polynomial interpolation. Besides that we substantially contributed to the two units on equivalence relations and partitions by providing the underlying formalizations, on whose basis the teaching of formal aspects in that area was organized. In each of these units, the invention of interactive elements fostering the students' intuition about the presented concepts was done on an exemplary basis, meaning the for each learning goal an individual interaction pattern has been derived and then programmed in a unique fashion. Reusing parts form one component inside another is practically impossible, except for the general layout and style, where we aimed at following a consistent paradigm throughout the entire system (e.g. the separation into "top area" containing input fields, sliders, and buttons and "bottom area" with a dynamic graphical representation reacting on the user settings given in the

top area). Differently the components giving prove support, where we provide the prove panel that only needs configuration in order to be reused in different examples.

Papers 10 through 12 show the persistent efforts to bring more *formal aspects of mathematics into the area of teaching mathematics* at various levels ranging from high school to graduate university math. With the commonly accepted computer algebra systems there is a strong tendency to over-emphasize algorithmic mathematics at the costs of a thorough formal treatment of abstract mathematical concepts and their underlying logical structure. In addition, the increasing popularity of dynamic geometry systems in particular among teachers leaves even less room for formal mathematics in the classroom.

The novel interactive interface components developed in this work not only proved useful in the learning units they were designed for. The feedback from test users gave us valuable insights that materialized in the interface design of Theorema 2.

Monographs #13 and #14: Algorithmische Methoden, Birkhäuser-Springer, 2008/2012. The monographs "Algorithmische Methoden" have been published in the frame of the Birkhäuser-Springer series "Mathematik kompakt". They are written in German because "Mathematik kompakt" is Birkhäuser's product for the then emerging bachelor programs in mathematics in German speaking countries in Europe. They are targeted towards students of mathematics in their first two semesters. We aim at bridging the gap between the traditionally well-established big blocks in every math curriculum—Lineare Algebra and Analysis—by presenting selected topics from both areas with an emphasis on algorithmic aspects and thorough discussions on how mathematical assistant systems can be used in these areas as well as on foundational aspects relevant for the development of systems supporting these fields.

The material covers exactly the content of the course "Algorithmic Methods" for beginning students of mathematics at JKU Linz. In its original design the course was based on the presentation of mathematical domains using the algorithmic version of predicate logic available in the Theorema system, see also the concrete example presented in Paper 7. Although desirable this frame could not be maintained for a broader audience, because of the limited popularity of mathematical assistant systems among potential teachers at other universities all over Europe. Due to this shift in paradigm the course has been completely redesigned and is since then based on the integrated discussion of both symbolic and numeric approaches to algorithmic mathematics, thus establishing a link between the areas of "Symbolic Computation" and "Numerical Mathematics"[10]. The mathematical assistant system aspect is now reflected in the use of widely spread mathematical software, namely Mathematica and Matlab for the symbolic and numeric parts of the course, respectively.

In its new form the course also picked up entirely new didactic concepts. Where in its original form the guideline was to formulate mathematical definitions and procedures in a language such that they can be executed immediately

[10]It was a big effort of the Special Research Program "Numerical and Symbolic Scientific Computing" (SFB F013) over a decade at JKU Linz to bring together these two branches of algorithmic mathematics. Both the installation of the course "Algorithmic Mathematics" as a mandatory course in the JKU math curriculum but even more this book project are visible offsprings from this cooperation.

without translation to a foreign program language, the new approach focuses on the aspect of *algorithm invention*. Although we reuse some of the ideas developed in Papers 10 to 12 for providing algorithm ideas when presenting the material in class, the main message propagated in the books is to *routinely extract algorithms from proofs*. For every algorithm discussed

- we start by giving a *precise specification of the problem* to be solved, then
- formulate a theorem expressing the existence (and if possible uniqueness) of a solution for the specification, proceed with a
- *constructive proof of the existence theorem*, and finally
- *turn the construction of the solution* to be read off the proof *into an algorithm* for computing the solution.

It has been frequently observed that the invention of new algorithms is a big hurdle for students, not only at beginner's level. Algorithms can be taught and their execution does usually not cause big issues, probably also because students are well trained in algorithm execution from high school mathematics. For the development of new problem solution strategies, however, they lack proper techniques they could apply, except in situations where the problem statement itself is given in procedural form.

Due to the predominance of imperative/procedural programming paradigms in the introductory programming courses, we often observe procedural thinking in mathematics even at the stage of problem specification. Very often students specify a mathematical problem not by *stating a property* that the desired solution has to fulfill but by *describing a procedure* how they aim to find the solution. While we do not want to condemn and abandon procedural thinking from mathematics entirely, we try to emphasize that this approach may sometimes fail. We propagate a strategy of using, as often as possible, induction proofs in order to first derive recursive algorithms. If necessary for efficiency reasons, recursions can then be turned into loops in a second step. This translation, however, we do not base on existing algorithms for conversion of certain types of recursion into loops as done in modern compilers. Instead we rely on intuitive inspection of recursion traces in order to obtain the corresponding loop performance.

In the topic selection we tried to balance between algorithms important from the point of view of symbolics as well as numerics. In many cases we manage to present *both* a symbolic approach *and* a numeric alternative. As a basis, volume 1 starts with basic arithmetic in integers, rationals, and real numbers. In order to be able to discuss symbolic algorithms later, we introduce the ideas of arithmetic with arbitrarily long integers and rational numbers at the very beginning. For having a fundament for numerics in the sequel, we go into floating point arithmetic as well. We then introduce vector spaces and univariate polynomials, in both deriving their basic mathematical properties followed by an in-depth discussion of how to appropriately represent the objects on a computer, key algorithms in the domain (scalar product, polynomial evaluation, and polynomial division), and conclude with the solution of one major problem in the domain (orthogonalization of vectors and polynomial interpolation).

Volume 2 covers functions in one variable, matrices, multivariate polynomials, and functions in several variables. The structure in each chapter follows the same principles like those in volume 1. The central problem we address in

every chapter of this volume is solving equations. Hence, the chapter on functions in one variable aims at solving equations in one variable using the Newton algorithm. As a basis for this, we elaborate symbolic function representations and algorithms for symbolic and numeric differentiation. The chapter on matrices treats fast matrix multiplication, matrix factorization, and solving systems of linear equations by the Gaussian algorithm. The chapter on multivariate polynomials is centered around the Gröbner bases method. We introduce polynomial reduction as a generalization of polynomial division introduced in volume 1, and present techniques for obtaining symbolic solutions of systems of algebraic equations. Finally, the chapter on functions in several variables treats systems of non-linear equations using the multivariate Newton method and non-linear least squares problems.

The novel contribution of these two monographs is of course not the mathematical content itself in a narrow sense. The topic selection and the didactic frame, however, reflect a completely new interpretation of algorithmic mathematics. The integrated view of symbolic and numerical mathematics with a uniform setup over eight chapters in two volumes that fulfills the requirements for both a serious treatment of numerics as well as an early introduction of symbolics to the students. Just as a simple example, in numerics a "polynomial" is usually regarded as a function (from reals into reals), whereas in symbolics we speak about polynomials as algebraic objects (certain sequences of coefficients) with certain operations, like polynomial division or factoring. The "polynomial function" associated with the polynomial is tightly connected but still something completely different. The identification of a unifying frame, in which both worlds can live together without noteworthy restrictions is certainly new. The tight integration of the material with the use of mathematical assistant systems is a new aspect in didactic literature. In particular this is true for the many aspects described in the books that do not deal with *just using* the algorithms provided by Mathematica and Matlab for solving the problems at hand. The main message of the books in this direction is a thorough treatment of the fundamentals for *implementing* the algorithms in a mathematical assistant system. Both volumes are written in theory exploration style so that there is a good chance to use them as a basis for a future formalization of the covered topics in Theorema 2.

CHAPTER II

PUBLICATIONS

1 *Refereed Publications Making Up the Habilitation Thesis*

This section consists of the publications referred to in Chapter I, except for #13 and #14, which are books published by Birkhäuser-Springer. We do not provide copies of the two monographs inside this volume but attach published copies as separate constituents of the thesis.

#1 W. Windsteiger. An Automated Prover for Zermelo-Fraenkel Set Theory in Theorema. *JSC*, 41(3-4):435–470, 2006. URL `http://authors.elsevier.com/sd/article/S0747717105001495`.

#2 B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger. Theorema: Towards Computer-Aided Mathematical Theory Exploration. *Journal of Applied Logic*, 4 (4):470–504, 2006. doi: http://dx.doi.org/10.1016/j.jal.2005.10.006.

#3 W. Windsteiger, B. Buchberger, and M. Rosenkranz. Theorema. In F. Wiedijk, editor, *The Seventeen Provers of the World*, volume 3600 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 96–107. Springer Berlin Heidelberg New York, 2006. ISBN 3-540-30704-4. URL `http://link.springer.com/book/10.1007/11542384`.

#4 E. M. Clarke, A. S. Gavlovski, K. Sutner, and W. Windsteiger. Analytica V: Towards the Mordell-Weil Theorem. In A. Bigatti and S. Ranise, editors, *Proceedings of Calculemus'06*, pages 35–50, 2006.[1]

[1] A note on publication data: Since it was originally planned for the proceedings to appear as a volume in *Electronic Notes in Theoretical Computer Science (ENTCS)* the paper was required to be written using the ENTCS-LaTeX-class that prints the "Published by Elsevier Science B. V." at the bottom of the titlepage. The paper was accepted in the category "Full Paper" but the official ENTCS volume did unfortunately not appear.

#5 W. Windsteiger. Theorema 2.0: A Graphical User Interface for a Mathematical Assistant System. In C. Kaliszyk and C. Lüth, editors, *Proceedings 10th International Workshop On User Interfaces for Theorem Provers, Bremen, Germany, July 11th 2012*, volume 118 of *Electronic Proceedings in Theoretical Computer Science*, pages 72–82. Open Publishing Association, 2012. ISBN 2075-2180 (ISSN). doi: 10.4204/EPTCS.118.5. URL `http://arxiv.org/abs/1307.1945v1`.

#6 W. Windsteiger. On a Solution of the Mutilated Checkerboard Problem using the Theorema Set Theory Prover. In S. Linton and R. Sebastiani, editors, *Proceedings of the Calculemus 2001 Symposium*, pages 28–47, 2001.[2]

#7 W. Windsteiger. Exploring an Algorithm for Polynomial Interpolation in the Theorema System. In T. Hardin and R. Rioboo, editors, *Proceedings of the Calculemus 2003 Symposium*, pages 130–136, Rome Italy, September 2003. Aracne Editrice S.R.L. ISBN 88-7999-545-6.

#8 M. Kerber, C. Rowat, and W. Windsteiger. Using Theorema in the Formalization of Theoretical Economics. In J. H. Davenport, W. M. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, volume 6824 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 58–73. Springer, 2011. ISBN 0302-9743 (ISSN). URL `http://dx.doi.org/10.1007/978-3-642-22673-1_5`.

#9 C. Lange, M. B. Caminati, M. Kerber, T. Mossakowski, C. Rowat, M. Wenzel, and W. Windsteiger. A Qualitative Comparison of the Suitability of Four Theorem Provers for Basic Auction Theory. In J. Carette, editor, *Conference on Intelligent Computer Mathematics (CICM 2013)*, volume 7961 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 200–215. Springer, 2013. ISBN 978-3-642-39319-8.

#10 G. Mayrhofer, S. Saminger, and W. Windsteiger. CreaComp: Computer-Supported Experiments and Automated Proving in Learning and Teaching Mathematics. In E. Milkova, editor, *Proceedings of ICTMT8*, 5 pages, 2007. ISBN 978-80-7041-285-5.

#11 G. Mayrhofer, S. Saminger, and W. Windsteiger. CreaComp: Experimental Formal Mathematics for the Classroom. In S. Li, D. Wang, and J.-Z. Zhang, editors, *Symbolic Computation and Education*, pages 94–114, Singapore, New Jersey, 2007. World Scientific Publishing Co. ISBN 978-981-277-599-3. URL `http://www.worldscibooks.com/socialsci/6642.html`.

#12 W. Windsteiger. Stimulating Students' Creativity Through Computer-Supported Experiments and Automated Theorem Proving. In E. Velikova and A. Andzans, editors, *Promoting Creativity for all Students in Mathematics Education*, pages 351–357, ISBN 978-954-712-420-2. Proceedings of Discussion Group 9, the 11th International Congress on Mathematical Education (ICME 11), Monterrey, Mexico, 2008.

#13 P. Kügler and W. Windsteiger. *Algorithmische Methoden – Zahlen, Vektoren, Polynome*. Reihe: Mathematik kompakt. Birkhäuser Basel Boston Berlin, 1st edition, December 2008. ISBN 978-3-7643-8434-0. URL `http://www.risc.uni-linz.ac.at/publications/books/AlgorithmischeMethoden/`.

#14 P. Kügler and W. Windsteiger. *Algorithmische Methoden – Funktionen, Matrizen, Multivariate Polynome*. Reihe: Mathematik kompakt. Birkhäuser Basel Boston Berlin, 1st edition, May 2012. ISBN 978-3-7643-8515-6. `www.risc.jku.at/publications/books/AlgorithmischeMethoden/`.

[2] A note on publication data: Submissions to Calculemus workshops/conferences have always been refereed seriously. The paper was accepted in the category "Full Paper" but the 2001 proceedings did not appear in the form of archival proceedings.

Available online at www.sciencedirect.com
SCIENCE DIRECT®
Journal of Symbolic Computation 41 (2006) 435–470
Journal of Symbolic Computation
www.elsevier.com/locate/jsc

An automated prover for Zermelo–Fraenkel set theory in *Theorema*☆

Wolfgang Windsteiger*

RISC Institute, A-4232 Hagenberg, Austria

Received 19 February 2003; accepted 5 April 2005
Available online 28 October 2005

Abstract

This paper presents some fundamental aspects of the design and the implementation of an automated prover for Zermelo–Fraenkel set theory within the *Theorema* system. The method applies the "Prove–Compute–Solve" paradigm as its major strategy for generating proofs in a natural style for statements involving constructs from set theory.

Keywords: Automated theorem proving; Set theory; Theorema

1. Introduction

The set theory prover in *Theorema* adapts the "**P**rove–**C**ompute–**S**olve" (for short: PCS) proving strategy for proofs containing language constructs from set theory. The PCS paradigm was introduced originally in Buchberger (2001) and it has already been applied successfully for proofs in elementary analysis in Vasaru-Dupré (2000). The main strategy in a PCS-oriented prover is to structure the proof generation into phases of

- proving (P), i.e. using inference rules for propositional connectives, the standard quantifiers from predicate logic, and for *theory-specific language constructs*,
- computing (C), i.e. rewriting w.r.t. formulae in the knowledge base,

☆ This work has been supported by "SFB Numerical and Symbolic Scientific Computing" (F013) at the University of Linz and the European Union "CALCULEMUS Project" (HPRN–CT–2000–00102).
* Tel.: +43 732 2468 9960; fax: +43 732 2468 9930.
E-mail address: Wolfgang.Windsteiger@RISC.Uni-Linz.ac.at.

0747-7171/$ - see front matter
doi:10.1016/j.jsc.2005.04.013

- solving (S), i.e. finding appropriate instances of existential variables occurring in the proof goal.

In general, of course, unification serves as the key method for solving formulae in arbitrary theories. For special theories, however, specialized solving techniques may be required and special techniques for solving from computer algebra may be applied. In particular, for solving formulae involving polynomial expressions there are powerful computer algebra methods such as the Gröbner bases method for systems of algebraic equations, see Buchberger (1985), or Collins' cylindrical algebraic decomposition method for systems of polynomial inequalities over the real closed fields, see Collins (1975). Having the computer algebra system Mathematica in the background of *Theorema*, we aim at applying these methods during the S-phase of our provers. In the context of set theory, solving can also lead to "solving for sets", i.e. finding sets that fulfill certain properties, but the set theory prover in its current state does not yet fully cover this aspect.

Many mathematicians are used to building up their theories in the frame of set theory, hence, computer support for doing proofs involving language constructs from set theory is a basic ingredient for computer-supported mathematics. The *Theorema* set theory prover aims to be primarily an educational tool that can support proving at an (under-)graduate university level in arbitrary theories that are built up in the frame of set theory. For this type of application it is important to incorporate also other proving techniques apart from sole set theory, notably arithmetic simplification in basic number domains or computational simplification involving finite sets. Moreover, since the *Theorema* syntax offers commonly used language constructs from set theory and the computational aspect of (finite) sets has always been supported in the *Theorema* computation environment, the set theory prover significantly enlarges the domain of applications for the *Theorema* system, because with this prover the *Theorema* system now offers *integrated support for both proving and computing* using set theory. It is important to mention that this prover is *not* intended as an automated prover for *proving set theory itself* based on only the axioms. Rather, it should be a tool that supports automated generation of "elegant proofs" in arbitrary mathematical theories that involve set theory.

Following the philosophy of most of the *Theorema* provers, the set theory prover aims at generating automated proofs in human-like natural style. In our experience, for mathematicians the acceptance of machine-generated proofs depends heavily on the readability of the proof for a human. In the automated theorem proving community, however, this aspect has not played a major role for a long time. Of course, as long as one does not display the proof, one can expand set-theoretic language constructs into first-order predicate logic and then apply powerful first-order theorem provers, like Otter, Vampire, or SPASS. The *Theorema* set theory prover, on the other hand, implements proof strategies applied by humans in an attempt to generate machine-proofs in a style acceptable by a human. Among other things, this will have considerable impact on computer-aided math education, which we currently see as one of the application areas for the *Theorema* system. It is attractive for the teacher to compose material in a mathematical language that is at the same time suitable for rigorous formal proofs and for execution of mathematical algorithms. It is attractive for the students to be able to perform computations immediately without translating mathematics into a programming language in order to execute the algorithms and to have certain proofs generated automatically in their lecture notes being able to do their own experiments. Of course, both the didactical potential and the dangers of such systems at certain levels of maths education need to be studied separately.

The current design of provers in the *Theorema* system requires a so-called "user prover" to be composed from "special provers", see Tomuta (1998). A special prover consists of a collection

of inference rules, whereas the user prover guides *the strategy*, through which the proof search procedure applies the inference rules. The P–C–S structure of the set theory prover is reflected in the composition of the set theory user prover from several special provers implementing the 'P', 'C', and 'S' phase, respectively. It consists of a *set theory proving unit* handling set-theory-related connectives and quantifiers in the goal or in the knowledge base, a *set theory computing unit* responsible for rewriting and simplification, and a *set theory solving unit* capable of instantiating existential goals resulting from unfolding definitions for set operations. In addition to these set theory specific components, the set theory prover re-uses several special provers already available in the *Theorema* system.

The description is structured as follows: Section 2 describes the theoretical basis upon which the set theory prover is built, Section 3 explains the interplay between user prover and special provers and gives an overview of the theorem proving procedure used in the *Theorema* system, Section 5 introduces the set theory proving units `STP` and `STKBR`, Section 6 describes the set theory computing unit `STC`, Section 7 presents the set theory solving unit `STS`, and finally we conclude with some examples of proofs generated by the set theory prover in Section 8.

2. The theoretical basis of the set theory prover

2.1. Set theory in the Theorema system

The use of "set theory" in the *Theorema* system is not tied to one particular axiomatization of set theory. Instead, a syntax for "sets" is introduced on the level of the "*Theorema* expression language", we refer to Windsteiger (2001a) for a detailed description of the language layers in the *Theorema* system. The language supports sets by providing *the braces* '{' and '}' as a flexible arity matchfix function symbol used for constructing finite sets, *the set quantifier* as a means for describing sets by a characteristic property, and several other language constructs commonly used in mathematics, such as e.g. '$\subseteq$', '$\cup$', '$\cap$', or '$\setminus$', see Kriftner (1998) for an overview on supported set syntax. By this, expressions such as $\{a, b, c\}$, $\{x \mid_x P_x\}$, $\{T_x \mid_x P_x\}$, $A \subseteq B$, $A \cup B$, $A \cap B$, or $A \setminus B$ are *syntactically valid expressions* in the *Theorema* language. In other words, the syntax of *Theorema* allows so-called "naive set theory", see Halmos (1960). However, the *Theorema* language does not fix a semantics for all set expressions supported in its syntax.

Semantics is attached to expressions in *Theorema* on the "inference rule level". The meaning of expressions is defined by inference rules that describe *how* certain operations on expressions can be performed. As an example, an inference rule for set expressions tells that in order to prove $x \in A \cap B$ we need to prove both $x \in A$ and $x \in B$. These inference rules are the elementary building blocks for provers within the *Theorema* system, see Section 3 for the details. The *intended semantics* of *Theorema* expressions is again that of naive set theory, i.e. $\{1, 4, 7\}$ is the proposed syntax for "the set containing exactly the elements 1, 4, and 7", $\{x \mid_x x < 10\}$ is meant to denote "the set of *all* x satisfying $x < 10$", $\{x^2 \mid_x x < 10\}$ is intended to mean "the set of *all* x^2, when x satisfies $x < 10$", $A \cap B$ should stand for "the intersection of A and B" and the like.

The *Theorema* language construct that deserves closer inspection in this context is the so-called *set quantifier*, which can appear in two variants $\{x \mid_x P_x\}$ and more generally $\{T_x \mid_x P_x\}$. In its first form, the set quantifier allows us to define a set from a "characteristic property" P_x. In the literature, this is often addressed as *set comprehension* or as *the abstraction* of a set

from a property and it goes back to G. Cantor, the founder of modern set theory. Naive set theory allows *unrestricted abstraction*, i.e. *for every* formula P_x the expression $\{x \mid_x P_x\}$ denotes the set of all x satisfying P_x, in combination with an *intuitive notion of membership*, namely $x \in \{x \mid_x P_x\} \Longleftrightarrow P_x$. Although intuitively "reasonable", this is the main drawback of naive set theory, since this is the main source for contradictions derivable in naive set theory such as the well known Russell paradox: We define $R := \{x \mid_x x \notin x\}$ and by straightforward reasoning on "membership" in the above mentioned intuitive sense we can quickly derive the contradiction $R \in R \Leftrightarrow R \notin R$.

Since naive set theory is inconsistent, it is not suitable as a theoretical basis for the inference rules used in our set theory prover. Hence, some *axiomatic set theory* must serve as the underlying theory for our prover. In axiomatic set theory the existence of certain sets and appropriate membership rules are introduced via axioms. There are different axiomatizations of set theory that provide fundamentally different solutions how to avoid Russell's paradox (and others):

- Zermelo–Fraenkel set theory (ZF) restricts abstraction to what is called *separation*. Roughly, it requires the structure $x \in S \wedge Q$ for P_x in an abstraction $\{x \mid_x P_x\}$, which disallows constructions like R. We refer to Ebbinghaus (1979) and Shoenfield (1967) for detailed treatments of ZF.
- Von-Neumann–Gödel–Bernays' axiomatization (NGB) of set theory, see e.g. Bernays and Fraenkel (1968) and Quine (1963), distinguishes between sets and classes and allows the membership predicate only for sets. Russell's paradox is avoided by showing that R is not a set an therefore $R \in R$ is not a well-formed assertion.
- Russell himself introduced type theory, where membership is only allowed for sets of different type, see Russell and Whitehead (1910). $R \in R$ is not allowed on the grounds that R and R are not of different type.

2.2. Possible approaches for set theory proving in the frame of Theorema

Due to the inconsistency of naive set theory there *cannot be a prover* that supports the *entire language for set theory* offered in the *Theorema* syntax and, at the same time, follows the *intended semantics* of expressions in the *Theorema* language as described above. Similar to the different approaches to axiomatization, there are different choices for how to integrate set theory proving into the *Theorema* system:

- We restrict the language, for which the prover is applicable instead of trying to support the entire language available for set theory in *Theorema*.
- We adapt the semantics of membership and deviate slightly from the intuitive semantics of well-known language constructs.
- We introduce a *typing concept* into the *Theorema* system and obey the types in all set theoretic language constructs either immediately on the syntax-level or on the inference-rule level. This would, however, require a fundamental re-design of the entire system and we decided not to follow this path for the moment.

It seemed most attractive to go for the first variant, thus we decided to choose ZF as the theoretical frame for the *Theorema* set theory prover. In particular, this choice was also motivated by the fact that evidently ZF is very popular among mathematicians and the principal target users of our

prover are mathematicians who want to formulate some mathematical theory using the language of set theory. In other words, the *Theorema* set theory prover does not support all of what the *Theorema* language syntax offers for set theory; it supports just that fragment of the *Theorema* language, which can safely be used according to the axioms of ZF as described in Section 2.3.

2.3. Zermelo–Fraenkel set theory as used in the set theory prover

ZF is an axiom system that guarantees the existence of certain sets. Based on these axioms, several new functions and predicates useful for set theory can then be introduced by explicit definitions. In the following, we will list those axioms and definitions from ZF, on which the inference rules of the set theory prover rely. At the same time, this section will describe exactly the fragment of the *Theorema* language that is actually supported by the set theory prover. Furthermore, we introduce some convenient abbreviations for commonly used formulations in set theory that are compatible with our prover. The *Theorema* set theory prover should, thus, be a useful tool for mathematicians embedding their work in some variant of ZF set theory that is consistent with these axioms, definitions, and abbreviations.

As already indicated above, the main challenge in an axiomatization of set theory is the definition of membership in a set described by the set quantifier. We now give the axioms of ZF forming the basis for those inference rules in the set theory prover that are applied for membership in expressions involving the set quantifier.

Axioms 2.1 (*Separation Axioms*). For every formula[1] P_x and every S, s.t. x is not contained in S and S is not contained in P_x, we have an axiom

$$\underset{z}{\exists}\,\underset{x}{\forall}\; x \in z \Longleftrightarrow x \in S \wedge P_x.$$

In the literature, the separation axioms are sometimes referred to as "subset axioms". They allow us—for any formula P_x and any term S (fulfilling the side-conditions given in Axiom 2.1)—to define "the set containing all x of S such that P_x", see Shoenfield (1967, pp. 239). In the *Theorema* syntax this set can be denoted as $\{x \underset{x\in S}{|} P_x\}$ or $\{x \in S \mid P_x\}$. From Axiom 2.1 we get

$$\underset{x}{\forall}\,(x \in \{x \underset{x\in S}{|} P_x\} \Longleftrightarrow x \in S \wedge P_x). \tag{1}$$

Axioms 2.2 (*Replacement Axioms*). For every formula Q_x and every S, s.t. S is not contained in Q_x, we have an axiom

$$\underset{x}{\forall}\,\underset{z}{\exists}\,\underset{y}{\forall}\,(y \in z \Leftrightarrow Q_x) \Longrightarrow \underset{z}{\exists}\,\underset{y}{\forall}\,(\underset{x\in S}{\exists} Q_x \Rightarrow y \in z).$$

In common mathematical practice, some special instances of the replacement axioms play a crucial role, namely for Q_x and S s.t. S is not contained in Q_x and Q_x has the form $P_x \wedge y = T_x$ for some formula P_x and some term T_x. For these special cases the respective replacement axioms justify the definition of "the set of all T_x when $x \in S$ satisfying P_x". The *Theorema* syntax for this set is $\{T_x \underset{x\in S}{|} P_x\}$ and, as shown in detail in Shoenfield (1967, pp. 240), from Axiom 2.2 we get

$$\underset{y}{\forall}\,(y \in \{T_x \underset{x\in S}{|} P_x\} \Longleftrightarrow \underset{x\in S}{\exists} P_x \wedge y = T_x). \tag{2}$$

[1] P_x indicates that the variable x occurs *free* in P_x. The expression P_x may contain other free variables than x as well.

At first sight, the construct $\{x \underset{x\in S}{|} P_x\}$ appears to be just a special case of $\{T_x \underset{x\in S}{|} P_x\}$, just take $T_x = x$. However, both the separation axioms and the replacement axioms are needed for the existence proof of $\{T_x \underset{x\in S}{|} P_x\}$, see Shoenfield (1967), thus, the separation axioms cannot be omitted.

The formulae (1) and (2) now define membership for special variants—note the required property $x \in S$—of the *Theorema* set quantifier as it can safely be used in ZF. The inference rules for membership as used in our set theory prover are, thus, based on (1) and (2). Once having the set quantifier, elementary set theory can be built up by just explicit definitions. For "sets" using variants of the set quantifier different from those shown in (1) and (2) ZF provides additional axioms guaranteeing their existence, see e.g. Shoenfield (1967).

From now on, if not stated otherwise, we want to use P, Q, R, and C as typed variables on the meta-level to denote *formulae*, all other letters shall denote *terms*. Free variables in formulae or terms will be indicated by subscripts. As long as the existence of the sets $\{x \underset{x}{|} P_x\}$ and $\{T_x \underset{x}{|} P_x\}$ is guaranteed by some axiom, we generalize (1) and (2) as follows:

$$\underset{x}{\forall}\,(x \in \{x \underset{x}{|} P_x\} \Longleftrightarrow P_x) \tag{3}$$

$$\underset{y}{\forall}\,(y \in \{T_x \underset{x}{|} P_x\} \Longleftrightarrow \underset{x}{\exists}\, P_x \wedge y = T_x). \tag{4}$$

Note that by this generalization we are now back at the naive set theory notion of membership for sets whose existence is guaranteed by ZF. Membership as in (4) is supported even in the more general case of a multiple range that binds more than one variable simultaneously. The multiple range in the set quantifier translates literally to the respective multiple range in the existential quantifier, i.e.

$$\underset{y}{\forall}\,(y \in \{T_{x_1,\ldots,x_n} \underset{x_1,\ldots,x_n}{|} P_{x_1,\ldots,x_n}\} \Longleftrightarrow \underset{x_1,\ldots,x_n}{\exists}\, P_{x_1,\ldots,x_n} \wedge y = T_{x_1,\ldots,x_n}).$$

It is convenient to allow also an additional condition in the set quantifier. We follow the convention to use for arbitrary range $\mathbf{x}$

$$\{\ldots \underset{\substack{\mathbf{x}\\ C}}{|} P\} \tag{5}$$

as an abbreviation for

$$\{\ldots \underset{\mathbf{x}}{|} C \wedge P\}. \tag{6}$$

Note, however, that we do not generalize the inference rules in the set theory prover to cover set quantifiers with conditions, we rather convert any expression of the form (5) in the goal or in the knowledge base into the corresponding form (6), before formulae are actually passed to the prover.

Definition 2.1 (*Subset, Set Equality*).

$$S^{(1)} \subseteq S^{(2)} :\Longleftrightarrow \underset{x}{\forall}\,(x \in S^{(1)} \Rightarrow x \in S^{(2)}) \tag{7}$$

$$S^{(1)} = S^{(2)} :\Longleftrightarrow \underset{x}{\forall}\,(x \in S^{(1)} \Leftrightarrow x \in S^{(2)}). \tag{8}$$

Definition 2.2 (*Empty Set, Set Difference*).

$$\emptyset := \{x \mid_x x \neq x\} \tag{9}$$

$$S^{(1)} \setminus S^{(2)} := \{x \mid_x x \in S^{(1)} \wedge x \notin S^{(2)}\}. \tag{10}$$

Definition 2.3 (*Finite Set Construction*). For any $n \geq 1$:

$$\{S^{(1)}, \ldots, S^{(n)}\} := \{x \mid_x x = S^{(1)} \vee \ldots \vee x = S^{(n)}\}. \tag{11}$$

Definition 2.4 (*Union, Intersection, Product*). For any $n \geq 2$:

$$S^{(1)} \cup \ldots \cup S^{(n)} := \{x \mid_x x \in S^{(1)} \vee \ldots \vee x \in S^{(n)}\} \tag{12}$$

$$S^{(1)} \cap \ldots \cap S^{(n)} := \{x \mid_x x \in S^{(1)} \wedge \ldots \wedge x \in S^{(n)}\} \tag{13}$$

$$S^{(1)} \times \ldots \times S^{(n)} := \{\langle x_1, \ldots, x_n\rangle \mid_{x_1,\ldots,x_n} x_1 \in S^{(1)} \wedge \ldots \wedge x_n \in S^{(n)}\}. \tag{14}$$

The notion $\langle \ldots \rangle$ is used for finite tuples provided as basic data type in *Theorema*. We do not model tuples *within* set theory but we use built-in knowledge about tuples provided by the semantics of the *Theorema* language.

Definition 2.5 (*Union, Intersection, Power Set*).

$$\bigcup S := \{x \mid_x \exists_{s\in S} x \in s\} \tag{15}$$

$$\bigcap S := \{x \mid_x \forall_{s\in S} x \in s\} \tag{16}$$

$$\mathcal{P}[S] := \{x \mid_x x \subseteq S\}. \tag{17}$$

Frequently used combinations of $\bigcup$ and $\bigcap$ with the set quantifier can conveniently be abbreviated when introducing $\bigcup$ and $\bigcap$ as quantifiers.

$$\bigcup_{\substack{x\in I\\ C_x}} S_x \quad \text{abbreviates} \quad \bigcup\{S_x \mid_{x\in I} C_x\} \tag{18}$$

$$\bigcap_{\substack{x\in I\\ C_x}} S_x \quad \text{abbreviates} \quad \bigcap\{S_x \mid_{x\in I} C_x\}. \tag{19}$$

When using the *Theorema* set theory prover one accepts the above definitions and assumes an underlying axiomatic system such as ZF that guarantees the existence of all these sets. We do not invent a new set theory that promises to be better suited for automated theorem proving, an approach that is taken elsewhere, e.g. in Formisano (2000).

3. How provers are organized in *Theorema*

3.1. Preliminaries on terminology

We will use the following terminology: a *proof situation* $K \vdash G$ is made up from a knowledge base of assumptions K and a goal G, and it should be understood as an abbreviation for the

phrase: "We have to prove G from K". Typically, the goal will be a single formula of the *Theorema* language, whereas the knowledge base consists of a collection of formulae, called the assumptions.

The task of the *special provers* is essentially the execution of individual *proof steps* that *reduce* the proof situation towards *terminal proof situations*, from which proof success or failure can easily read off. Terminal proof situations will be denoted by just their "value", e.g. 'proved' or 'failed'. The rules applied by the special provers guiding the reduction of proof situations are called *inference rules*. Thus, an inference rule turns a proof situation $K \vdash G$ into a proof situation $K' \vdash G'$ with a new goal G' and a new knowledge base K'. In the description of inference rules, we will denote an inference rule named 'I' transforming $K \vdash G$ into $K' \vdash G'$ by

$$\mathbf{I}: \frac{K' \vdash G'}{K \vdash G}$$

(read as: "The rule 'I' justifies a proof step to reduce the proof of G from K to a proof of G' from K'"). This notation is similar to notations used in logic for describing inference rules in formal proof calculi (e.g. the natural deduction calculus or the Gentzen calculus). Certain similarities to these formalisms are desired, but we use it purely as a symbolic description for proof steps, and we do not refer to any meaning of the symbols in any known logic system.

We give an example of a well-known inference rule from the natural deduction calculus for predicate logic written in this style:

$$\textbf{ArbitraryButFixed}: \frac{K \vdash P_{x \to x_0}}{K \vdash \underset{x}{\forall} P_x} \quad \text{(where } x_0 \text{ is a new constant)}.$$

The rule 'ArbitraryButFixed' tells us that, in order to prove $\underset{x}{\forall} P_x$ (from K) it suffices to prove $P_{x \to x_0}$ (from K) for a new constant x_0, where $P_{x \to x_0}$ stands for "P with each free occurrence of x substituted by x_0".

3.2. The generation of proofs in Theorema

The automated generation of proofs in the *Theorema* system is based on three main components: the user prover, the special provers, and the global proof search procedure.

3.2.1. User provers

The *Theorema* user interface provides the command

$$\texttt{Prove}[G,\ \text{using} \to K,\ \text{by} \to M],$$

which initiates an attempt to prove the goal G using the knowledge base K by the method M. In the *Theorema* terminology, we call the available prove-methods *user provers*. A user prover is a program that sets up a particular two-row configuration grid (see Fig. 1) of special provers and then passes control to *Theorema*'s *global proof search procedure*.

3.2.2. Special provers

A *special prover* is a sequential collection of inference rules. In Fig. 1 the sequential structure of a special prover is visualized by a top-to-bottom line-up of the inference rules in each of

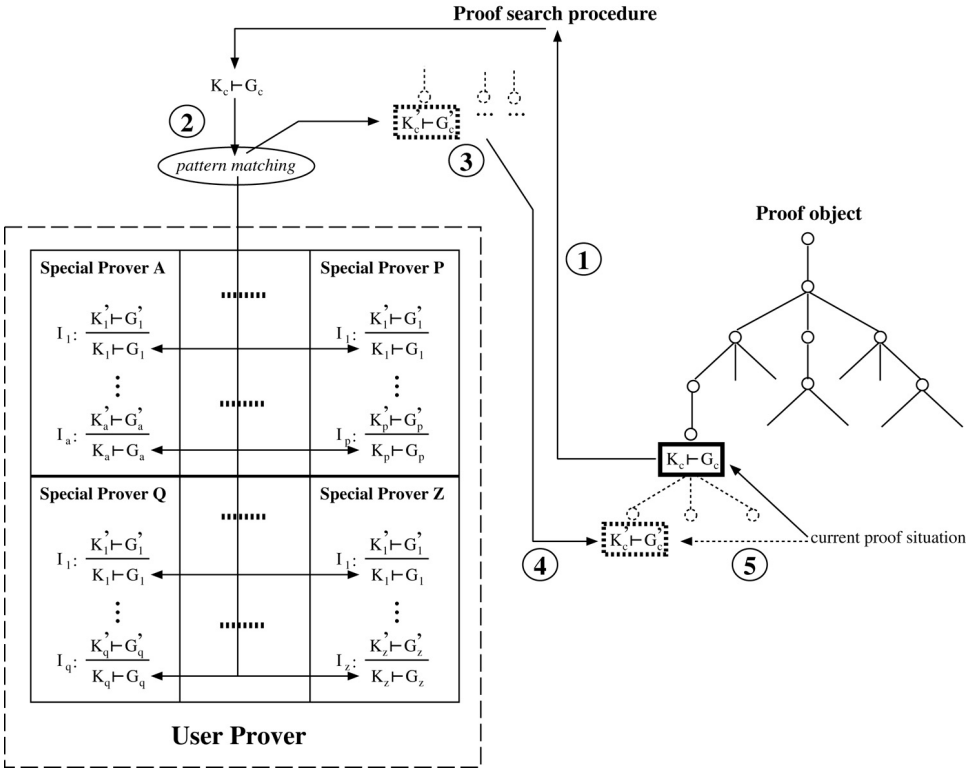


Fig. 1. Proof search and prover composition.

the special provers. The arrangement of the special provers in the user prover's configuration grid results essentially in a *hierarchical collection of inference rules* structured by the placement within the two-dimensional grid and, on a finer level, by the sequential arrangement within each cell in the grid.

3.2.3. The global proof search procedure and the proof object

The proof search procedure uses inference rules as arranged in the user prover's configuration grid in order to manipulate the *global proof object*. The proof object has a tree structure with each node containing one proof situation. The proof search procedure maintains a *current proof situation*, which specifies the node that is to be manipulated next. Initially, the proof object consists of only the root node containing the initial proof situation $K \vdash G$ given by the user in the `Prove`-command. Each proof tree manipulation is the augmentation of the proof object at the current proof situation by one or more new nodes, whose contents depend on the inference rules found by the proof search procedure in the special provers. Fig. 1 visualizes the main phases of one proof step:

1. The proof search procedure extracts the current proof situation $K_c \vdash G_c$ from the proof object.
2. Mathematica's pattern matching mechanism is used to select appropriate inference rules that allow to reduce the current proof situation. Inference rules are implemented as Mathematica programs taking goal, knowledge base, and "additional facts" of the current proof situation as input. We refer to Section 5 for more details on the role of the "additional facts" in a

proof situation. As output, an inference rule returns a new node to be inserted into the proof tree. In this phase, the configuration grid in the user prover set-up is crucial: The special provers in the first row are tried left to right, in each prover the rules are tested top to bottom. The *first rule* whose goal and assumptions match the current goal G_c and the current knowledge base K_c will be selected. If none of the special provers in the first row applies, the special provers in the second row are tried again left to right. In each special prover the rules are tried again top to bottom, and *from each applicable prover*, the first rule matching the current proof situation will also be selected. Note that the pattern language of Mathematica contains conditionals. Thus, the selection of inference rules based on Mathematica's pattern matching is not restricted to purely syntactical matches but it allows also the test of certain conditions.

3. All inference rules selected in the previous phase will be applied in this proof step to the current proof situation resulting in new nodes to be inserted into the proof object.
4. If there is more than one new node, each node is assigned a new branch in the proof object. Branches reflect *alternative proof attempts* in a proof.
5. Finally, the current proof situation is stepped to the new node on the leftmost new branch.
6. These steps are iterated until a terminal proof situation 'proved' or the search depth limit is reached. If a proof fails on one branch by reaching either a terminal proof situation 'failed' or the maximal search depth then the proof search continues on the next branch. Once all branches have failed, the entire proof has failed.

For details on the organization of the proof search within *Theorema* we refer to Tomuta (1998).

The implementation of the user prover arranges the special provers in the grid and the implementations of the special provers arrange the inference rules within the special provers. Thus, the experience of the prover programmer is reflected in a smart set-up of the user prover and the special provers. The proof search as described above is completely automated with no possibility for user-interaction. As an alternative, the *Theorema* system offers also an interactive proof search mechanism, see Piroi (2004).

4. The *Theorema* set theory prover

As discussed in Section 3.2 the implementation of a prover for set theory must consist of a user prover and several special provers. In the following, 'set theory prover' will refer to the user prover for set theory, which combines newly developed special provers for set theory with previously developed general purpose special provers available in the *Theorema* system, such as `TerminalND` for detecting terminal proof situations, `BasicND` and `PND` for basic and general predicate logic reasoning, `QR` for rewriting w.r.t. quantified equalities, equivalences, or implications in the knowledge base, or `CDP` for treatment of case distinctions, see Buchberger and Vasaru (2000), Vasaru-Dupré (2000) and Windsteiger (2001a). We will describe the four new special provers that have been developed for set theory.

`STP` collects inference rules with some set-theoretic symbol as the outermost symbol in the proof goal.

`STKBR` expands set-theoretic notions in the assumptions.

`STC` performs simplification by computation on finite sets.

`STS` applies special techniques for instantiation of existential formulae in the proof goal, which are useful in the context of set theory.

The set theory prover arranges `TerminalND`, `STKBR`, `STC`, `STP`, and `STS` in this order from left to right in the first row of the configuration grid and `BasicND`, `QR`, `CDP`, and `PND` in the second row. Roughly, this results in a heuristic to generally first expand set-theoretic language constructs in the knowledge base when they appear as outermost symbols before working on the proof goal. When reducing the proof goal, simplifications based on computational knowledge for finite sets are applied before expanding set-theoretic language constructs by their definition. If non-set-theory symbols appear as outermost symbols, proceed by the usual predicate logic reasoning and rewriting. The set theory prover can be used in interactive proving like all other provers in the *Theorema* system. The emphasis of this work is, however, to set up the prover for completely automated proof generation.

5. STP and STKBR: The set theory proving units

The PCS proof strategy imposes a structure on proofs as alternating phases of proving, computing, and solving, as already described in Section 1. Inference rules for set theory specific *proving* are provided in the two new special provers `STP` and `STKBR`. During the Prove-phase, we alternate steps of *reducing the goal* with steps of *expanding the knowledge base*. While `STP` reduces set theory specific language constructs in the proof goal, `STKBR` expands them in the knowledge base. The set theory prover arranges both special provers in the first row of the configuration grid.

5.1. Set theory specific goal reduction

Set theory specific goal reduction is implemented as a special prover named `STP` (for `Set Theory Proving`). The inference rules within `STP` differ mainly in the syntactic patterns for the proof situation. A few inference rules are influenced in addition by global variables, by which, for instance, certain inference rules can be deactivated. Some strategies depend on the proof progress stored in `STP`'s *local proof context*, which is part of the "additional facts"-parameter in the implementation of inference rules, see Section 3.

The inference rules are grouped into rules for *membership*, rules for *inclusion*, and rules for *set equality*. The rules for membership cover proof situations, where the outermost symbol in the proof goal is '$\in$'. There is at least one inference rule for each "kind of set" introduced in Section 2, in some cases we provide specialized rules in order to offer special treatment for special cases. We show some of the membership rules as they are used in `STP`.

$$\textbf{MembershipSeparation}: \frac{K \vdash t \in S \wedge P_{x\to t}}{K \vdash t \in \{x \mid_{x\in S} P_x\}}$$

The inference rule 'MembershipSeparation' is just a reformulation of variant (1) of the separation Axioms 2.1. Hence, its correctness is an immediate consequence of (1) and we do not give a separate correctness proof for this rule. In fact, most of the rules in `STP` are just direct translations of one of the axioms or one of the definitions listed in Section 2. For these cases we do not give the correctness proofs of the inference rules used in our prover. Some of the inference rules, however, condense several inference steps into one compact rule to be applied. In these cases, we provide hand-proofs for the correctness of the respective rules. An example of such a rule is the elimination of the union-quantifier in the goal. Simply using abbreviation (18) would lead to an inference rule

$$\textbf{Membership-U}: \frac{K \vdash t \in \bigcup\{S_x \mid_{x\in s} C_x\}}{K \vdash t \in \bigcup\limits_{\substack{x\in s\\ C_x}} S_x}$$

The STP prover, however, implements the rule

$$\textbf{MembershipUnionOf}: \frac{K \vdash \exists\limits_{x\in s}\,(t \in S_x \wedge C_x)}{K \vdash t \in \bigcup\limits_{\substack{x\in s\\ C_x}} S_x}$$

'MembershipUnionOf' reduces the proof of $t \in \bigcup\limits_{\substack{x\in s\\ C_x}} S_x$ to prove $\exists\limits_{x\in s}\,(t \in S_x \wedge C_x)$.

Proof. Assume $\exists\limits_{x\in s}\,(t \in S_x \wedge C_x)$, thus $t \in S_{x_0} \wedge C_{x_0}$ for some constant $x_0 \in s$. With $z := S_{x_0}$ we can infer from this $t \in z \wedge C_{x_0} \wedge z = S_{x_0}$, hence

$$\exists\limits_{z}\,(\exists\limits_{x\in s}\, t \in z \wedge C_x \wedge z = S_x). \tag{20}$$

Separating the quantifiers in (20) gives $\exists\limits_{z}(t \in z \wedge \exists\limits_{x\in s}\,(C_x \wedge z = S_x))$, which, by (2), is equivalent to $\exists\limits_{z}\,(t \in z \wedge z \in \{S_x \mid_{x\in s} C_x\})$. By (15) this is equivalent to $t \in \bigcup\{S_x \mid_{x\in s} C_x\}$, thus $t \in \bigcup\limits_{\substack{x\in s\\ C_x}} S_x$ by (18). □

As special rules for membership in finite sets and finite unions, respectively, we provide for $n \geq 2$

$$\textbf{MembershipFinite}: \frac{t \neq S^{(2)}, \ldots, t \neq S^{(n)}, K \vdash t = S^{(1)}}{K \vdash t \in \{S^{(1)}, S^{(2)}, \ldots, S^{(n)}\}}$$

$$\textbf{MembershipUnion}: \frac{t \notin S^{(2)}, \ldots, t \notin S^{(n)}, K \vdash t \in S^{(1)}}{K \vdash t \in \bigcup\{S^{(1)}, S^{(2)}, \ldots, S^{(n)}\}}$$

Again, we omit the easy proofs of correctness. Note, that both membership in finite sets and finite unions would reduce by definition to a disjunction of formulae. The majority of human mathematicians would proceed by a smart choice of one of the alternatives and then just prove the chosen alternative. The inference rules given above, however, reduce the proof of a disjunction further to the proof of always the first alternative while assuming the negations of the remaining alternatives. This has the advantage that the rule keeps the proof search space small because it does not introduce additional branches into the proof object. On the other hand, the resulting proofs appear "unnatural" for human mathematicians at the point where these rules are applied. Thus, these rules might be adapted in future versions of the prover.

As we will see in Section 6, these rules will not be applied in the standard set-up of the set theory prover, because as soon as the set theory computation unit is present, membership in finite sets and finite unions will be decided based on computational semantics available in the *Theorema* language. The two rules are contained in STP only because, in the spirit of modular system design, we do not presuppose that all future *Theorema* user provers combine the available

special provers for set theory in exactly the way in which they are combined in our set theory prover now.

Furthermore, we provide one general inference rule for set inclusion and set equality, respectively, which reduce inclusion and equality to membership according to Definition 2.1. Additionally, we provide special rules for special cases in order to reduce the search depth in the proof search procedure, like e.g.

$$\textbf{ConjunctionSubset}: \frac{\text{proved}}{K \vdash \{x \mid_{x} \ldots \wedge x \in S \wedge \ldots\} \subseteq S}$$

$$\textbf{SubsetSeparation}: \frac{P_{x\to x_0}, x_0 \in X, K \vdash x_0 \in Y \wedge Q_{y\to x_0}}{K \vdash \{x \mid_{x\in X} P_x\} \subseteq \{y \mid_{y\in Y} Q_y\}}$$

(where x_0 is some new constant).

For the empty set, the expansion of Definition 2.2 would result in "unnatural" proof steps, hence, we provide special rules for the empty set, like e.g.

$$\textbf{EmptySetSubset}: \frac{\text{proved}}{K \vdash \emptyset \subseteq S}$$

$$\textbf{EqualsEmptySet}: \frac{K \vdash \neg P_{x\to x_0}}{K \vdash \{T_x \mid_{x} P_x\} = \emptyset}$$

(where x_0 is some new constant). The proofs for these rules are again straightforward. For more details and a *complete listing of all inference rules* used in STP we refer to Windsteiger (2001a).

5.2. Set theory specific knowledge expansion

5.2.1. Knowledge expansion by lazy level saturation

The special prover STKBR (for Set Theory Knowledge Base Rewriting) uses "lazy saturation" in order to infer *new knowledge* from formulae already contained in the knowledge base using knowledge about set theory specific language constructs. In contrast to classical level saturation methods, which try to obtain *all formulae* that can be inferred from the knowledge base in one saturation run, "lazy saturation" is somewhat more moderate in that it only finds formulae that can be inferred from the original knowledge base at the beginning of the saturation run. This has the advantage that usually less "potentially unimportant" formulae are generated before the prover continues with some other steps. However, if no other proof steps can be performed, the proof search procedure will continue with subsequent lazy saturation steps. This type of "iterated lazy saturation" *can* ultimately lead to a completely saturated knowledge base, but unlike classical saturation techniques it *does not necessarily*.

The STKBR prover is implemented as just *one inference rule* implementing the mechanism of lazy saturation based on knowledge from set theory combined with simplification of assumptions based on computational semantics from the *Theorema* language. STKBR is considered to be applicable to the current proof situation as soon as new formulae occur in the knowledge base compared to its previous application. This check is done with the help of an entry in the local proof context passed among the "additional facts" of the current proof situation, see Section 3. Similar to STP, most of the set theory specific knowledge expansion rules used in this phase

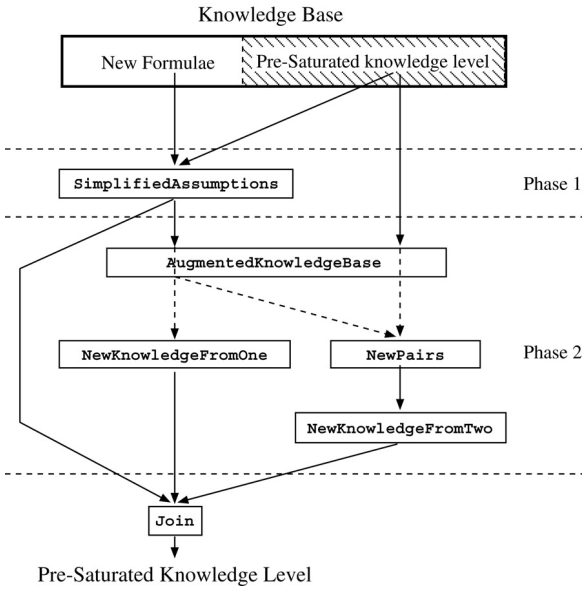


Fig. 2. Schematic flow of lazy level saturation as used in STKBR.

are just re-formulations of the axioms and the definitions in Section 2. Again they are grouped into rules for membership, inclusion, and set equality. For each "kind of set" we provide a membership rule for a proof situation, where '$\in$' appears as outermost symbol in one of the assumptions. Moreover, the prover contains rules for unfolding membership inside universally quantified formulae and, like STP, some special rules based on elementary set theory knowledge. We refer to the examples in Section 8.2.2 for more details on these special rules.

Knowledge expansion in STKBR happens in two phases:

1. New formulae are simplified using built-in semantic knowledge available in the *Theorema* language semantics.
2. New knowledge is *inferred* from the simplified knowledge base by lazy saturation as described above. The expansion rules used in lazy saturation are grouped into two groups:
   - *Group* I containing rules for inferring new knowledge from *one* known formula and
   - *Group* II containing rules for inferring new knowledge from *two* known formulae.

   Matching rules from Group I are applied to the simplified new formulae, matching rules from Group II are applied to all new pairs of formulae containing at least one new formula.

All formulae generated during these two phases are added to the knowledge base and, finally, the formula labels of formulae contained in the expanded knowledge base are stored in the local proof context in order to be accessible in the next saturation run. We call this stage a *pre-saturated knowledge level*. Complete level saturation would iterate this process until no more new formulae can be inferred. Lazy saturation, instead, passes control back to the proof search procedure after one iteration.

The schematized flow of STKBR's level saturation mechanism is shown in Fig. 2. Phase 1 is accomplished by applying the function 'SimplifiedAssumptions' to two arguments: the entire knowledge base and a list of labels 'sat' describing the pre-saturated knowledge level from a previous saturation run. Each new formula from the knowledge base is sent through a simplification function, which *computes* a simplified version of the formula w.r.t. semantic

knowledge from the *Theorema* language. In fact, the same simplification function is used that is applied also in the STC module for *goal simplification by computation* and in the top-level *Theorema*-user command Compute, see Section 6. This guarantees utmost coherence between all computations happening in different components of the *Theorema* system, be it on the user level by calling Compute, be it on the prover level by doing simplifications on the goal or on the knowledge base. Formulae that cannot be simplified as well as formulae from the previous saturation level leave phase 1 unchanged. Actually, STKBR contributes to both the P- and the C-phase, hence, it is not a pure proving unit! We allowed this mixture of P- and C-phase in *one* special prover in the current implementation merely for reasons of efficiency.

Phase 2 is covered in the implementation by the function 'AugmentedKnowledgeBase', which receives the simplified knowledge base resulting from phase 1 and again the list 'sat'. 'NewKnowledgeFromOne' applies Group I of expansion rules to *each (simplified) new assumption*, 'NewKnowledgeFromTwo' applies Group II of expansion rules to *all new pairs* that can be formed using at least one new assumption. The new formulae obtained in phase 2 joined with the simplified knowledge base resulting from phase 1 give the new pre-saturated knowledge level.

5.2.2. Rule locking

Rule locking is a mechanism that helps to prevent cycles in the proof search during level saturation. As an example, consider the two inference rules

$$\mathbf{I}: \frac{\ldots, x \in A, x \in B, \ldots \ \vdash \ G}{\ldots, x \in A \cap B, \ldots \ \vdash \ G} \qquad \mathbf{I'}: \frac{\ldots, x \in A \cap B, \ldots \ \vdash \ G}{\ldots, x \in A, \ldots, x \in B, \ldots \ \vdash \ G}$$

occurring in STKBR. We call two rules *inverse to each other* if one rule neutralizes the effect of the other. **I** and **I'** are an example of rules being inverse to each other. Our prover contains some pairs of inverse rules although, generally, we try to avoid to provide inverse rules wherever possible. Inverse rules need special attention because their unrestricted use immediately results in a cycle in the proof search. Rule locking allows us to dynamically disable certain inference rules for certain values of the input parameters. In the above example, the application of rule **I'** resulting in a new formula F will automatically prevent rule **I** from being applied to F in the remainder of this proof branch. Similarly, the application of rule **I** producing new formulae F_1 and F_2 will block rule **I'** on F_1 and F_2 in the remainder of this proof branch. Note, however, that affected rules are only locked for *particular values of the input parameters* whereas they stay applicable in all other situations.

In general, for each pair of inverse rules **I** and **I'** we implement both **I** and **I'** such that they lock their inverse for certain inputs. There is no general law, however, for which inputs a rule must be locked. As a special case, inference rules may even lock *themselves* in order to avoid "uninteresting" expansions in the proof search. Consider again the example from above: Applying rule **I'** once would add the new assumption $x \in A \cap B$. During the next saturation run, **I'** would add the new assumptions $x \in A \cap (A \cap B)$ and $x \in B \cap (A \cap B)$ and so forth. Thus, rule **I'** is implemented such that it locks both itself and also its inverse rule **I** on the new formulae generated by **I'**. Rule locking utilizes STKBR's local proof context to store this type of information on the proof progress.

6. STC: The set theory computing unit

The *Theorema* language contains semantics essentially for *finite sets*, namely

- sets that are constructed using the set braces '{' and '}' as *set constructor* applied to finitely many arguments, and
- sets that are constructed using *algorithmic versions* of the set quantifiers introduced in Section 2, see also Buchberger (1996), i.e. set quantifiers with finite and computable range specifications, see Windsteiger (2001a). In particular, *integer ranges* and *set ranges with finite sets* are algorithmic ranges, which lead to finite sets when used in combination with the set quantifiers.

The *Theorema* language semantics allows the *explicit construction* of finite sets as an enumeration of the (finitely many) elements contained in the set, i.e. the language contains some data-structure representing a finite set. Set operations (such as union, intersection, power set, etc.) on finite sets are implemented as operations on the data-structure for finite sets in a constructive fashion, i.e. every operation on finite sets results again in a finite set. Tests for membership, inclusion, or set equality for finite sets, thus, reduces to *testing finitely many cases*, which is implemented in the frame of the *Theorema* language semantics as well.

Computation using built-in semantics knowledge is available in the *Theorema* system through the top-level user command `Compute`. A typical computation involving finite sets and numbers is

$$\texttt{Compute}[\{3x \mid_{x\in\{1,2,3,4\}} \text{is-prime}[x]\},\ \text{built-in}\rightarrow\langle\text{Built-in}[\text{“Sets”}],\ \text{Built-in}[\text{“Numbers”}]\rangle]$$

resulting in the finite set {6, 9} and, of course,

$$\texttt{Compute}[6 \in \{3x \mid_{x\in\{1,2,3,4\}} \text{is-prime}[x]\},\ \text{built-in}\rightarrow\langle\text{Built-in}[\text{“Sets”}],\ \text{Built-in}[\text{“Numbers”}]\rangle]$$

results in *True*. Internally, `Compute` sends the expression to be computed to a simplification function, which simplifies the expression with respect to both *user-defined knowledge* given in the "using"-option and *built-in knowledge* from the *Theorema* language semantics given in the "built-in"-option to `Compute`. In the examples above, no user-defined knowledge is provided and built-in knowledge about "Sets" (for the set quantifier and the finite set in the range of the quantifier) and "Numbers" (for 'is-prime' and for the multiplication used in '$3x$' is applied. It is the intention of the `STC` (for `S`et `T`heory `C`omputing) special prover to integrate the computational power available for finite sets seamlessly into the *Theorema* proving machinery. Otherwise, all algorithmic knowledge about finite sets needs to be re-implemented inside the set theory prover, which would make it very difficult to guarantee consistent behavior in proving and computing. In order to avoid this duplication of code and knowledge and in order to achieve coherent simplifications on the top-level using `Compute` and on the proving-level in simplifications of both the proof goal and the knowledge base, the `STC` prover simplifies the goal by sending the goal formula to the same simplification function that is also used in `Compute` and in `STKBR`.

Basically, when the `STC` prover applies to a proof situation, one proof step consists of calling the simplification function on the proof goal and, if the result differs from the original form, of adding a new node to the proof object, from which the effect and a complete trace of the computation can be displayed. In fact, the interface to the underlying simplification function is implemented in a more flexible fashion. Namely, it allows *arbitrary built-in knowledge* available in the *Theorema* language in addition to built-in knowledge about finite sets to be used during simplification. Similar to the `Compute`-examples above, the user may specify built-in knowledge in the call of any prover using the option "built-in". The set theory prover sets up the environment such that simplification uses set theory semantics by default and user-specified built-in semantics

in addition. Given a proof goal such as $6 \in \{3x \mid_{x\in\{1,2,3,4\}} \text{is-prime}[x]\}$ the set theory prover will produce different proof variants depending on the additional semantical knowledge provided by the user:

- Using *no semantic knowledge at all* by completely deactivating STC, the proof goal will be reduced by an inference rule from STP to prove

  $$\exists_{x\in\{1,2,3,4\}} \text{is-prime}[x] \wedge 6 = 3x,$$

  and, by predicate logic reasoning using *matching against formulae in the knowledge base*, this goal might eventually be proven since $x = 2$ is an appropriate choice for the existential variable. Note, however, that formulae such as 'is-prime[2]' and '$6 = 3 * 2$' must be provided *explicitly* in the knowledge base. Moreover, for the proof of subgoal $2 \in \{1, 2, 3, 4\}$, STP will apply the inference rule **MembershipFinite** described in Section 5.1 resulting in a very fine-grained proof showing all details down to the axioms of set theory.
- Using the standard setting with *only built-in sets*, the resulting proof is essentially the same, just that the subgoal $2 \in \{1, 2, 3, 4\}$ can be simplified to *True* by STC in one stroke. However, the set quantifier does not simplify to a finite set, since 'is-prime' is unknown and, thus, the expression stays unevaluated.
- Using *built-in semantics of "Sets" and semantics of 'is-prime'*, the proof goal will be simplified by STC to prove

  $$6 \in \{3 * 2, 3 * 3\},$$

  and again STP's **MembershipFinite** will come into play. Still, $6 = 3 * 2$ must be available in the knowledge base in order to finish the proof.
- Using *built-in semantics of "Sets" and "Numbers"*, the proof goal will be simplified by STC in one step to *True* and the proof succeeds without any additional explicit knowledge.

Of course, each of these variants has its pros and cons and the *Theorema* user can decide which path to follow. We consider it of utmost importance for practical applications and acceptance of the system to offer this choice to the user. More details on combining computation with proving can be found in Windsteiger (2001a). Furthermore, we refer to Section 8.2 for more examples of interaction of proving and simplification by computing.

7. STS: The set theory solving unit

The special prover STS (for Set Theory Solving) collects inference rules for eliminating existential quantifiers in the proof goal. Its name suggests that this prover deals with set theory specific aspects of solving, but, since general predicate logic solving components in the *Theorema* system are not yet far advanced, the STS prover in its current state collects inference rules for existential goals as they result typically from proof goals containing language constructs from set theory. Set theory specific solving in the sense of "solving for sets" meaning "finding sets that fulfill certain properties" is not yet dealt with in this version of the prover. Rather, we concentrate on using the special structure of the existential formula in order to devise efficient instantiation methods for certain types of existential goals.

The inference rules in STS mainly cover proof situations of the form $K \vdash \exists_x P_x$, i.e. we have to *find some term* t such that $K \vdash P_{x\to t}$. In many cases the choice of t can be made essentially

on the basis of K. In these situations, the methods used for instantiating the existential goal are *matching* parts of P_x against ground formulae in K and *unification* of parts of P_x with formulae in K.

The more difficult cases, however, are those, where the choice of t depends strongly on the inner structure of P_x. In these situations, the instantiation of the existential goal requires, roughly speaking, some further processing of P_x before an appropriate choice of t is possible. Hence, we *introduce a solve-constant* in place of the existential variable. A solve-constant differs from a Skolem-constant in that it is a placeholder for the term t, whose exact "form" is not yet known at the time when the solve-constant is introduced, whereas a Skolem-constant is a new constant about which we do not know anything. For the proof to succeed, *all solve-constants* that have been introduced must be expressed through appropriate *ground terms* in such a manner that the resulting formula can be proven. The introduction of solve-constants, thus, allows us to delay the instantiation of existential goal formulae to a later phase of the proof in order to be able to proceed with standard reasoning techniques before actually instantiating the existential variable. What we call a solve-constant is often addressed as a *meta-variable* by other authors. The technique of meta-variables is well known and used also in other systems.

Essentially, solve-constants imitate what a human often does when proving $\underset{x}{\exists}\ P_x$, namely to "pretend to know x" and then reason on P_x in order to derive more knowledge on x until we can identify a t that fulfills all the requirements collected for x. Of course, the strategy after introducing solve-constants must always be to isolate the solve-constants, and then to apply special solving techniques depending on the nature of the remaining formulae. Hence, this strategy reduces proving to *solving over various domains*. Ideally, the formulae to be solved are algebraic equalities or inequalities such that known solution techniques available from computer algebra can be applied for finding an appropriate t. For this reason some of the inference rules in `STS` employ the Mathematica `Solve`-function in situations where an existentially quantified variable or a solve-constant appears inside an equality. For requirements formed by arbitrary set-theoretic formulae we plan to develop an appropriate *solving calculus* as future work.

We present only one typical inference rule from `STS`.

$$\textbf{IntroSolveConstant}: \frac{K \quad \vdash \quad Q_{y\to y^*} \wedge \underset{x\in s}{\exists}\ (P_x \wedge y^* = T_x) \wedge R_{y\to y^*}}{K \quad \vdash \quad \underset{y}{\exists}\ (Q_y \wedge y \in \{T_x \mid_{x\in s} P_x\} \wedge R_y)}$$

where Q_y and R_y are possibly empty conjunctions of formulae and y^* is a solve-constant. The inference rule described above might appear random. It is part of `STS` since it applies exactly to proof situations left after expanding membership in special unions, namely goals of the form $m \in \bigcup\{T_x \mid_{x\in s} P_x\}$. It can be observed in many examples involving proof goals of this form (see in particular the example in Section 8.2.4) that this strategy leads to a well-structured proof. The rule eliminates the outermost existential quantifier by introducing a solve-constant and it introduces another existential quantifier by immediately expanding membership. `STS` contains further rules, which allow the elimination of the remaining existential quantifier in this particular case and even in other more general situations, see Windsteiger (2001a). Note, that the solve-constant already appears in an isolated position, so that it can immediately be expressed by the ground term T_x as soon as P_x has been solved for x. In addition to rules introducing solve-constants, the `STS` prover, of course, also contains several rules for instantiating solve-constants as soon as they appear in an isolated position. We refer also to the discussion in Section 8.2.4,

Table 1
Comparison to systems on examples from CASC-18

Example	*Theorema*	E-SETHEO	Vampire	DCTP	Bliksem	Saturate
SET010	3.0	15.8	23.9	1.2	>300	?
SET014	3.2	>300	>300	281.0	>300	1.8
SET096	2.0	9.6	17.0	113.7	7.1	8.1
SET171	4.0	>300	>300	>300	>300	2.9
SET580	8.7	0.4	0.1	1.5	>300	1.7
SET612	2.1	>300	>300	>300	>300	9.9
SET624	43.7	0.7	0.8	1.7	>300	10.2
SET630	2.4	0.4	62.3	1.5	>300	116.8
SET716	6.8	>300	>300	>300	>300	8.8

where, in particular, the important role of *solving as a sub-problem in proving* is discussed in a concrete example.

8. Comparison and examples

8.1. Comparison to state-of-the-art theorem provers

In this section, we test the *Theorema* set theory prover on some examples from the SET section of TPTP, see TPTP: Thousands of Problems for Theorem Provers (n.d.). Timings refer to CPU seconds consumed on a 1500 MHz Intel P4 running Mathematica 4.2 and include the time needed for generating the proof, simplifying the successful proof, and displaying the formatted proof as shown in Section 8.2. Table 1 shows a comparison of the computing times to state-of-the-art theorem provers as they performed in CASC-18[2], see CADE-18 ATP System Competition (CASC-18) (n.d.), which refer to CPU time on a 993 MHz Intel P4. The timings of the "Saturate"-prover were taken from Ganzinger and Stuber (2003) and were measured on a 2000 MHz CPU (timings are only available for examples from the FOF division (first-order form) of CASC-18).

The former winner of the FOF division of previous CASCs, SPASS, did not participate in CASC-18. Table 2 compares timings of the *Theorema* set theory prover on some of the SET examples contained in TPTP to the performance of a revised version of SPASS as reported in Afshordel et al. (2001). SPASS's timings have been recorded on a 333 MHz Intel P2, *Theorema* timings refer to experiments on a 400 MHz Intel P2.

We want to emphasize, however, that the absolute computation times are not our main focus in the current stage of development, mainly because the *Theorema* system is currently implemented in the programming language of Mathematica, which does not offer possibilities for compiling programs. Hence, comparing the run-time of *interpreted Mathematica code* to computing times of *optimized compiled machine code* does not tell us much. The timings in Tables 1 and 2 are meant to demonstrate that our set theory prover generates proofs "within a few seconds" even for examples where other provers fail completely or need considerably more time, see e.g. (SET014), (SET171), (SET612), or (SET716). The Saturate prover performs very well on the previously mentioned examples, but interestingly it is slower by a factor of almost 50 in (SET630). Note, on the other hand, that the *Theorema* set theory prover is considerably slower than the CASC

[2] Software versions used: E-SETHEO csp02, Vampire 5.0, DCTP 10.1p, Bliksem 1.12a.

Table 2
Theorema versus SPASS

Example	*Theorema*	SPASS 2.0
SET010	6.1	1
SET612	7.5	1
SET624	155.4	101
SET694	5.5	1
SET698	22.7	71
SET722	6.6	18
SET751	5.04	3

provers in (SET624), which will be discussed in detail in Section 8.2.5. Table 2 shows that the *Theorema* set theory prover and SPASS show "similar" behavior.

Of course, proof generation should be fast, but we are currently much more interested in having automatically generated "nice proofs" that are easily understandable for a human reader. We therefore aim at designing provers that apply inference rules in a smart and "natural human-like way" without too many failing branches during the proof search. Once this is achieved, the absolute computation times depend only on the efficiency of executing the programs on particular hardware. We can speed up the entire system (i.e. *all provers* available in the *Theorema* system!) by improving the runtime environment, on which the *Theorema* system is based. One possibility, which we are currently investigating for a re-design of *Theorema*, is to develop an efficient execution engine (based on e.g. Java) for a certain fragment of the Mathematica programming language that would allow the compilation of our provers. From first experiments an envisaged speed-up by a factor between 50 and 100 seems realistic. One can observe in practical examples as shown in the subsequent sections that the proofs generated by our set theory prover contain only a *few failing branches*, and each branch contains only a *few useless formulae*.

8.2. Proofs generated by the Theorema set theory prover

In this section, we collect some representative proofs that were generated *completely automatically* by the *Theorema* set theory prover. In order to justify our claim from Section 8.1 that "the set theory prover generates proofs that are easily understandable for a human reader", the examples in the subsequent sections will not only describe the methods and heuristics used in our set theory prover, but they will also include the generated proofs. The proofs are displayed in *simplified form*, i.e. they do not contain anymore failing proof branches and they do not show any formulae that did not contribute to the final proof success. The *fully automated simplification* of the "raw proof object" as it is produced by the set theory prover is a standard post-processing feature available in the *Theorema* system and the timings given in Section 8.1 include also the time needed for proof simplification.

The optical appearance of the proofs in the *Theorema* system corresponds exactly to how they are typeset in this paper! Within *Theorema*, the standard presentation of proofs is generated in a Mathematica notebook document, a document format provided by Mathematica that allows typeset mathematical text to be intermixed with Mathematica input and output expressions as well as graphics. The proofs have been translated from Mathematica notebook format into LaTeX as accurately as possible without manual beautification. Some of the features of the *Theorema* standard proof presentation utilize, however, special capabilities of the Mathematica notebook format and can therefore not be rendered in this paper:

- Formulae in the knowledge base and goal formulae are displayed in different color.
- Formula labels in running text are "click-able" and show the entire referenced formula in a popup-window when clicked.
- Proof branches are organized in a hierarchy of nested cells that reflects the structure of the proof. Collapsing entire proof-branches by mouse-click allows us to quickly browse through the structure of a proof and easily "zoom into" the interesting proof parts or skip uninteresting proof parts, respectively.

8.2.1. Properties of functions built into the set theory prover

The *Theorema* mathematical language supports the notion $f :: A \to B$ denoting the predicate "f is a function from A to B (in intensional form)". In intensional form, a function from A to B is something that can be applied to some term in A resulting in a term in B. Alternatively, *Theorema* offers the concept of a function from A to B in extensional form (written $f : A \to B$) from set theory, where a function is a certain subset of $A \times B$. As an example, we take (SET722), where the set theory prover succeeds for both intensional and extensional representation for functions. The computing times do not essentially differ between the two variants. We present the proof of (SET722) based on the intensional function concept, in order to demonstrate that the use of the set theory prover does not require the user to force all of mathematics into set representation.

The *Theorema* set theory prover does not require the definition of surjectivity in its knowledge base. Rather, it recognizes surjectivity on the inference rule level, i.e. the prover contains inference rules for proving surjectivity and for expanding surjectivity in the knowledge base, respectively, regardless of whether the intensional or the extensional function concept is used.

Proof (*SET722*). $\underset{A,B,C,f,g}{\forall} f :: A \to B \wedge g \circ f :: A \overset{surj.}{\to} C \Rightarrow g :: B \overset{surj.}{\to} C$,

under the assumption:

(Definition (Composition)) $\underset{f,g,x}{\forall} (g \circ f)[x] := g[f[x]]$.

We assume

(1) $f_0 :: A_0 \to B_0 \wedge g_0 \circ f_0 :: A_0 \overset{surj.}{\to} C_0$,

and show

(2) $g_0 :: B_0 \overset{surj.}{\to} C_0$.

In order to show surjectivity of g_0 in (2) we assume

(3) $x1_0 \in C_0$,

and show

(4) $\underset{B1}{\exists} B1 \in B_0 \wedge g_0[B1] = x1_0$.

From (1.1) we can infer

(6) $\underset{A1}{\forall} A1 \in A_0 \Rightarrow f_0[A1] \in B_0$.

From (1.2) we know by definition of "surjectivity"

(7) $\underset{A2}{\forall} A2 \in A_0 \Rightarrow (g_0 \circ f_0)[A2] \in C_0$,

(8) $\underset{x2}{\forall} x2 \in C_0 \Rightarrow \underset{A2}{\exists} A2 \in A_0 \wedge (g_0 \circ f_0)[A2] = x2$.

By (8), we can take an appropriate Skolem function such that

(9) $\underset{x2}{\forall} x2 \in C_0 \Rightarrow A2_0[x2] \in A_0 \wedge (g_0 \circ f_0)[A2_0[x2]] = x2$.

Formula (3), by (9), implies:

$A2_0[x1_0] \in A_0 \wedge (g_0 \circ f_0)[A2_0[x1_0]] = x1_0$,

which, by (6), implies:

$f_0[A2_0[x1_0]] \in B_0 \wedge (g_0 \circ f_0)[A2_0[x1_0]] = x1_0$,

which, by (Definition (Composition)), implies:

(10) $f_0[A2_0[x1_0]] \in B_0 \wedge g_0[\, f_0[A2_0[x1_0]]] = x1_0$.

Formula (4) is proven because, with $B1 := f_0[A2_0[x1_0]]$, (10) is an instance. □

The use of the special inference rules for function properties like e.g. surjectivity can be suppressed by an option in the Prove-call. The knowledge base would then need to contain the definition of surjectivity explicitly. The proof of (SET722), however, succeeds even in this setting. It differs only in that the special inference rule combines several proof steps into one compact step. Special inference rules are available also for injectivity, which are used in (SET716), where the proof takes just 6.8 s, which is about the same time that the "Saturate"-prover needs. Note, however, that all the CASC provers fail in (SET716).

8.2.2. Set theory specific knowledge built into the prover

The examples in this section shall demonstrate how set theory specific knowledge is built into the prover. As already mentioned in the description of the individual special provers, most of the inference rules used in the set theory prover are in essence only re-formulations of the *axioms and definitions of ZF*. Some rules it uses, however, are based *on certain theorems* that are valid in ZF. Clearly, these theorems are themselves only consequences of the axioms, therefore *all inference rules in the set theory prover are based on the axioms of ZF*. What we want to say is that there are certain rules that correspond to direct application of an axiom and there are other rules that hide a chain of logical arguments based on the axioms. The examples in this section try to show that this is reasonable because the *Theorema* set theory prover is intended for mathematicians who want to have support in their every-day work using sets. It is not intended to be a prover that reduces every mathematical proof to the axioms of ZF.

Proof (*Proposition (Intersection Powerset*)). $\bigcap \mathcal{P}[A] = \emptyset$,

with no assumptions.
We have to prove (Proposition (intersection powerset)), hence, we have to show:

(1) $A1_0 \notin \bigcap \mathcal{P}[A]$.

We prove (1) by contradiction.
We assume

(2) $A1_0 \in \bigcap \mathcal{P}[A]$,

and show (a contradiction).
From (2) we can infer

(3) $\underset{A2}{\forall} A2 \in \mathcal{P}[A] \Rightarrow A1_0 \in A2$.

From (3) we can infer

(4) $A1_0 \in \emptyset$,

(5) $A1_0 \in A$.

Using available computation rules we can simplify the knowledge base:
Formula (4) simplifies to

(6) *False*.

Formula (a contradiction) is true because the assumption (6) is false. □

It can be proven in ZF that the power set $\mathcal{P}[A]$ always contains $\emptyset$ and A itself, and, of course, the *Theorema* set theory prover can prove this. Thus, we can instantiate a universally quantified variable running over the power set of A with $\emptyset$ and A. This theorem is coded into a special inference rule in STKBR, which allows the instantiation of the universally quantified assumption (3) to infer (4) and (5). The simplification of (4) to (6) is then accomplished in phase 1 of the subsequent saturation run in STKBR by built-in semantic knowledge about finite sets (in particular, the empty set) as described in Section 5.2.

The second example is taken from the case study on the mutilated checkerboard, see McCarthy (1964, 1995) and Windsteiger (2001b,a). The theorem says that an 8×8 checkerboard with two opposite corners missing can always be covered by dominos. A proof of this theorem can be given using a formulation of the problem in set theory. A proof of the theorem has been generated using *Theorema* by building up the theory of dominos, checkerboards, coverings, etc. and by completely exploring new notions as they are defined. "Completely exploring", in this context, means that sufficiently many properties of a new notion are proven before the next notion will be introduced, see Buchberger (1999). Although each of the proofs in this exploration is generated completely automatically, the whole proof cannot be called "fully automated", because the exploration itself is the interaction of the user with the *Theorema* system. This is a highly non-trivial, mathematically very interesting and challenging, and didactically very instructive experience for the human user. Using systems such as *Theorema*, the mathematician can then concentrate on this task of structuring mathematical knowledge in big theories, while the actual proofs are then assisted by the computer. For students this opens the possibility to experiment on building up own theories and to, in the optimal case, *understand why* certain known theories are built up in a certain way.

During one of these so-called exploration rounds, we arrived at the proposition that whenever X is a domino on the board then the domino covers two distinct fields that are adjacent to each other.

Proof (*Proposition (Dominos Adjacent)*).

$$\underset{X}{\forall} \text{domino-on-board}[X] \Rightarrow X \subseteq \text{Board} \wedge \underset{x,y}{\exists} x \in X \wedge y \in X \wedge x \neq y \wedge \text{adjacent}[x, y],$$

under the assumption:

(Definition (Domino))

$$\underset{x}{\forall}\ (\text{domino-on-board}[x] :\Leftrightarrow x \subseteq \text{Board} \wedge |x| = 2 \wedge$$

$$\underset{x1,x2}{\forall}\ x1 \in x \wedge x2 \in x \wedge x1 \neq x2 \Rightarrow \text{adjacent}[x1, x2]).$$

We assume

(2) $\text{domino-on-board}[X_0]$,

and show

(3) $X_0 \subseteq \text{Board} \wedge \underset{x,y}{\exists}\ x \in X_0 \wedge y \in X_0 \wedge x \neq y \wedge \text{adjacent}[x, y]$.

Proof of (3.1) $X_0 \subseteq$ *Board*: (SKIPPED)
Proof of (3.2):
Formula (2), by (Definition (Domino)), implies:

(5) $|X_0| = 2 \wedge X_0 \subseteq \mathit{Board} \wedge$

$$\underset{x1,x2}{\forall}\ x1 \in X_0 \wedge x2 \in X_0 \wedge x1 \neq x2 \Rightarrow \text{adjacent}[x1, x2].$$

From (5.1) we can infer

(6) $X1_0 \in X_0$,

(7) $X1_1 \in X_0$,

(8) $X1_0 \neq X1_1$.

Now, let $y := X1_0$. Thus, for proving (3.2) it is sufficient to prove:

(10) $\underset{x}{\exists}\ x \in X_0 \wedge X1_0 \in X_0 \wedge x \neq X1_0 \wedge \text{adjacent}[x, X1_0]$.

Now, let $x := X1_1$. Thus, for proving (10) it is sufficient to prove:

(15) $X1_1 \in X_0 \wedge X1_0 \in X_0 \wedge X1_1 \neq X1_0 \wedge \text{adjacent}[X1_1, X1_0]$.

Using available computation rules we evaluate (15) using (8) and (5.1) as additional assumption(s) for simplification:

(16) $X1_1 \in X_0 \wedge X1_0 \in X_0 \wedge \text{adjacent}[X1_1, X1_0]$.

Proof of (16.1) $X1_1 \in X_0$:
Formula (16.1) is true because it is identical to (7).
Proof of (16.2) $X1_0 \in X_0$:
Formula (16.2) is true because it is identical to (6).
Proof of (16.3) $\text{adjacent}[X1_1, X1_0]$:
Formula (16.3), using (5.3), is implied by:

(17) $X1_0 \in X_0 \wedge X1_1 \in X_0 \wedge X1_1 \neq X1_0$.

Using available computation rules we evaluate (17) using (8) and (5.1) as additional assumption(s) for simplification:

(18) $X1_0 \in X_0 \wedge X1_1 \in X_0$.

Proof of (18.1) $X1_0 \in X_0$:
Formula (18.1) is true because it is identical to (6).
Proof of (18.2) $X1_1 \in X_0$:
Formula (18.2) is true because it is identical to (7). □

In this proof, special knowledge about cardinality goes into the proof. `STKBR` uses an inference rule that allows us to choose distinct elements from a finite set, i.e. if we know $|A| = n$ for some natural number n then we can choose *new constants* $x_1, \ldots, x_n$ such that $x_i \in A$ (for each $1 \leq i \leq n$) and "all x_i distinct". This rule allows us to infer (6), (7), and (8) from (5.1) in the proof above, the remaining proof is straightforward.

8.2.3. The interplay between Theorema and Mathematica

The two proofs in this section will show, how *Theorema* interacts with the underlying Mathematica system. We want to emphasize the strict separation between *Theorema* and Mathematica in the sense that no Mathematica algorithm from the rich computer algebra library available through Mathematica is applied "quietly" during a proof in the *Theorema* system unless the user explicitly allows the *Theorema* set theory prover to do so. The first example uses Mathematica's `Solve` function for instantiating existential variables in the proof goal.

Proof (*SET751*).

$$\underset{A,B,f,X,Y}{\forall} X \subseteq A \wedge Y \subseteq A \wedge X \subseteq Y \wedge f :: A \to B \Rightarrow \text{image}[f, X] \subseteq \text{image}[f, Y],$$

under the assumption:

(Definition (Image)) $\underset{f,X}{\forall} \text{image}[f, X] := \{ f[x] \mid \underset{x}{} x \in X \}$.

We assume

(1) $X_0 \subseteq A_0 \wedge Y_0 \subseteq A_0 \wedge X_0 \subseteq Y_0 \wedge f_0 :: A_0 \to B_0$,

and show

(2) $\text{image}[f_0, X_0] \subseteq \text{image}[f_0, Y_0]$.

For proving (2) we choose

(3) $f1_0 \in \text{image}[f_0, X_0]$,

and show:

(4) $f1_0 \in \text{image}[f_0, Y_0]$.

From (1.3) we can infer

(8) $\underset{X2}{\forall} X2 \in X_0 \Rightarrow X2 \in Y_0$.

Formula (3), by (Definition (Image)), implies:

(11) $f1_0 \in \{ f_0[x] \mid_x x \in X_0 \}$.

From (11) we know by definition of $\left\{T_x \mid_x P\right\}$ that we can choose an appropriate value such that

(12) $x1_0 \in X_0$,

(13) $f1_0 = f_0[x1_0]$.

Formula (4), using (13), is implied by:

$f_0[x1_0] \in \text{image}[f_0, Y_0]$,

which, using (Definition (Image)), is implied by:

(19) $f_0[x1_0] \in \{ f_0[x] \mid_x x \in Y_0 \}$.

In order to prove (19) we have to show

(20) $\exists_x x \in Y_0 \wedge f_0[x1_0] = f_0[x]$.

Since $x := x1_0$ solves the equational part of (20) it suffices to show

(21) $x1_0 \in Y_0$.

Formula (21), using (8), is implied by:

(22) $x1_0 \in X_0$.

Formula (22) is true because it is identical to (12). □

Since a sub-formula of the existential goal (20) is an equality containing the existential variable, we instantiate the existential variable x in the proof goal with the help of Mathematica. In this example, a candidate for x was found by solving the equation $f_0[x1_0] = f_0[x]$ for x, which is done by a call to the Mathematica function `Solve` for solving (systems of) equations. Of course, unification or even matching would have done this job as well, but, in the case of equational sub-formulae, the `STS`-prover tries to apply the *specific rule* using `Solve` *before it tries general predicate logic solving* using matching and unification.

The second example shows, how arithmetic knowledge on natural numbers provided by Mathematica is accessible for the set theory prover. As already discussed in Section 6 on the set theory computation unit `STC`, semantic knowledge about natural numbers from the *Theorema* language is not available in the set theory prover by default, but it can be provided by the user on demand in the call of the prover using the "built-in"-option.

Proof (*G*). $36 \in \bigcup_{i \in \mathbb{N}} \{ j^2 \mid_{j \in \mathbb{N}} j \geq i \wedge j \leq i + 5 \}$

under the assumption

(A) $\forall_{m,n} n > m \Rightarrow \exists_i i \leq n \wedge i \geq m \wedge i \in \mathbb{N}$.

In order to show (G) we have to show

(1) $\exists_i 36 \in \{ j^2 \mid_{j \in \mathbb{N}} j \geq i \wedge j \leq i + 5 \} \wedge i \in \mathbb{N}$.

In order to prove (1) we have to show

(2) $\exists_i \exists_j\ j \geq i \wedge j \in \mathbb{N} \wedge j \leq i + 5 \wedge i \in \mathbb{N} \wedge 36 = j^2$.

Since $j := 6$ solves the equational part of (2) it suffices to show

(4) $\exists_i\ i \in \mathbb{N} \wedge 6 \geq i \wedge 6 \in \mathbb{N} \wedge 6 \leq 5 + i$.

Using available computation rules we evaluate (4):

(6) $\exists_i\ i \leq 6 \wedge i \geq 1 \wedge i \in \mathbb{N}$.

Formula (6), using (A), is implied by:

(7) $6 > 1$.

Using available computation rules we evaluate (7):

(8) *True*.

Formula (8) is true because it is the constant *True*. □

The derivations of formulae (1) and (2) result from applying STP inference rules for membership in a union and membership in a set abstraction, respectively. Reduction of (2) to (3) is accomplished by instantiating j by a solution of a quadratic equation done in STS. Similar to the previous example, since a sub-formula of the existential goal (2) is an equality containing the quantified variable, the Mathematica Solve function is used internally to solve the quadratic equation $36 = j^2$ for j, which finds two solutions $j = -6$ and $j = 6$. Of course, in this example matching and unification would not be an alternative, since theory-specific arithmetic knowledge is necessary for solving this formula. The first solution results in a failing proof attempt, since $-6 \in \mathbb{N}$ simplifies to *False* by built-in knowledge about $\mathbb{N}$. The failing branch is eliminated when finally simplifying the successful proof. Note that the labels of the formulae indicate a "missing branch". Formulae (3) and (5) do not appear in the proof presentation because they have been eliminated during proof simplification. Simplifications from (4) to (6) and from (7) to (8) were made using available semantic knowledge about natural numbers by STC ($6 \in \mathbb{N}$ and $6 > 1$, respectively) and, finally, reduction from (6) to (7) and the detection of proof success were made by standard predicate logic inference rules. We have no specialized solving methods for natural numbers available, therefore we needed assumption (A) in the knowledge base. An appropriate solver for $\mathbb{N}$ would be able to verify (6) without any additional knowledge. We will investigate necessary solving techniques in future work.

8.2.4. Theory exploration versus isolated theorem proving

We consider (SET770), an example from the TPTP library concerning equivalence classes, namely the theorem that two equivalence classes are equal or disjoint. Note again, that none of the provers in the CASC competition could solve this problem. In Windsteiger (2001a), an entire exploration of the theory of equivalence relations, equivalence classes, factor sets, partitions, induced relations, etc. is given. Instead of proving (SET770) from first principles, i.e. from the axioms, it is preferable to first prove some auxiliary lemmata, which later facilitate the proof of the theorem. This is just what a human mathematician would be doing. We present here the proof of (SET770) using the two auxiliary propositions (equal classes) and (not in distinct classes) in the knowledge base. The computing time for the proof is 5.8 s on a 2000 MHz Intel P4, the proofs

of the auxiliary propositions take 8.5 and 8.6 s, respectively. Of course, this example using the two additional propositions is not anymore (SET770) in the sense of TPTP! It should be clear that the timings for the examples given in Section 8.1 refer to the problem formulation as specified in the TPTP library, except that, of course, we may omit definitions in the knowledge base that refer to set theory specific constructs, which are covered by inference rules in our prover. We do not claim this example to be a "solution for (SET770) as given in TPTP" and, therefore, we also did not include it in the tables of timings in Section 8.1. We rather show this example in order to advocate for "theory exploration" being superior to "proving from first principles", in particular if we want mathematicians to appreciate our systems.

Proof (*SET770*). $\underset{R,x,y}{\forall}$ is-symmetric$[R] \wedge$ is-transitive$[R] \Rightarrow$

$(\text{class}[x, R] = \text{class}[y, R]) \vee (\text{class}[x, R]) \cap \text{class}[y, R] = \{\})$.

under the assumptions:

(Proposition (equal classes)) $\underset{R,x,y}{\forall}$ is-transitive$[R] \wedge$ is-symmetric$[R] \wedge$

$\langle x, y\rangle \in R \Rightarrow \text{class}[x, R] = \text{class}[y, R]$,

(Proposition (not in distinct classes)) $\underset{R,x,y,z}{\forall}$ is-symmetric$[R] \wedge$ is-transitive$[R] \wedge$

$x \in \text{class}[y, R] \wedge x \in \text{class}[z, R] \Rightarrow \langle y, z\rangle \in R$.

We assume

(1) is-symmetric$[R_0] \wedge$ is-transitive$[R_0]$,

and show

(2) $(\text{class}[x_0, R_0] = \text{class}[y_0, R_0]) \vee (\text{class}[x_0, R_0]) \cap \text{class}[y_0, R_0] = \{\})$.

We prove (2) by proving the first alternative negating the other(s).
We assume

(4) $\text{class}[x_0, R_0] \cap \text{class}[y_0, R_0] \neq \{\}$.

We now show

(3) $\text{class}[x_0, R_0] = \text{class}[y_0, R_0]$.

From (4) we know that we can choose an appropriate value such that

(5) $x3_0 \in \text{class}[x_0, R_0] \cap \text{class}[y_0, R_0]$.

From (5) we can infer

(7) $x3_0 \in \text{class}[x_0, R_0]$,

(8) $x3_0 \in \text{class}[y_0, R_0]$.

Formula (3), using (Proposition (equal classes)), is implied by:

(11) is-symmetric$[R_0] \wedge$ is-transitive$[R_0] \wedge \langle x_0, y_0\rangle \in R_0$.

Proof of (11.1) is-symmetric$[R_0]$:

Formula (11.1) is true because it is identical to (1.1).
Proof of (11.2) is-transitive[R_0]:
Formula (11.2) is true because it is identical to (1.2).
Proof of (11.3) $\langle x_0, y_0\rangle \in R_0$:
Formula (11.3), using (Proposition (not in distinct classes)), is implied by:

(12) $\underset{x}{\exists}$ is-symmetric[R_0] $\wedge$ is-transitive[R_0] $\wedge$ $x \in$ class[x_0, R_0] $\wedge$ $x \in$ class[y_0, R_0].

Now, let $x := x3_0$. Thus, for proving (12) it is sufficient to prove:

(13) is-symmetric[R_0] $\wedge$ is-transitive[R_0] $\wedge$ $x3_0 \in$ class[x_0, R_0] $\wedge$ $x3_0 \in$ class[y_0, R_0].

Proof of (13.1) is-symmetric[R_0]:
Formula (13.1) is true because it is identical to (1.1).
Proof of (13.2) is-transitive[R_0]:
Formula (13.2) is true because it is identical to (1.2).
Proof of (13.3) $x3_0 \in$ class[x_0, R_0]:
Formula (13.3) is true because it is identical to (7).
Proof of (13.4) $x3_0 \in$ class[y_0, R_0]:
Formula (13.4) is true because it is identical to (8). □

The same case study has been carried out for an intensional concept of relations. Similar to the intensional concept of a function described in Section 8.2.1, an intensional relation is something that can be applied to terms yielding true or false. An intensional relation is nothing else than a predicate in the sense of logic. We show one of the proofs and explain its key steps, since this proof shows the natural interplay between P-, C-, and S-phases as implemented in the *Theorema* set theory prover very nicely. This example also demonstrates that the PCS-strategy, which has already been used in a prover for elementary analysis within the *Theorema* system, see Buchberger (2001); Vasaru-Dupré (2000), yields natural proofs very similar to the style a human mathematician would give the proof.

Proof (*Lemma (Union Inverse Factor Set)*). $\underset{A}{\forall}$ is-reflexive$_A$[$\sim$] $\Rightarrow$ $\bigcup$ factor-set$_\sim$[A] $= A$,

under the assumptions:

(Definition (relation sets): class) $\underset{A,x}{\forall}$ class$_{A,\sim}$[x] $:= \{a \mid_a a \in A \wedge a \sim x\}$,

(Definition (relat. sets): factor-set) $\underset{A}{\forall}$ factor-set$_\sim$[A] $:= \{$class$_{A,\sim}$[x] $\mid_x x \in A\}$,

(Definition (reflexivity)) $\underset{A}{\forall}$ is-reflexive$_A$[$\sim$] $:\Leftrightarrow$ $\underset{x}{\forall}$ $(x \in A \Rightarrow x \sim x)$.

We assume

(1) is-reflexive$_{A_0}$[$\sim$],

and show

(2) $\bigcup$ factor-set$_\sim$[A_0] $= A_0$.

Formula (2), using (Definition (relation sets): factor-set), is implied by:

$$\bigcup\{\mathrm{class}_{A_0,\sim}[x] \mid_x x \in A_0\} = A_0,$$

which, using (Definition (relation sets): class), is implied by:

(3) $\bigcup\{\{a \mid_a a \in A_0 \wedge a \sim x\} \mid_x x \in A_0\} = A_0.$

Formula (1), by (Definition (reflexivity)), implies:

(4) $\forall_x (x \in A_0 \Rightarrow x \sim x).$

We show (3) by mutual inclusion:
$\subseteq$: We assume

(5) $x1_0 \in \bigcup\{\{a \mid_a a \in A_0 \wedge a \sim x\} \mid_x x \in A_0\}$

and show:

(6) $x1_0 \in A_0.$

From (5) we know by definition of the big $\bigcup$-operator that we can choose an appropriate value such that

(7) $x2_0 \in \{\{a \mid_a a \in A_0 \wedge a \sim x\} \mid_x x \in A_0\},$

(8) $x1_0 \in x2_0.$

From (7) we know by definition of $\{T_x \mid_x P\}$ that we can choose an appropriate value such that

(9) $a1_0 \in A_0,$

(10) $x2_0 = \{a \mid_a a \in A_0 \wedge a \sim a1_0\}.$

Formula (8), by (10), implies:

(23) $x1_0 \in \{a \mid_a a \in A_0 \wedge a \sim a1_0\}.$

From (23) we can infer

(24) $x1_0 \in A_0 \wedge x1_0 \sim a1_0.$

Formula (6) is true because it is identical to (24.1).
$\supseteq$: Now we assume

(6) $x1_0 \in A_0.$

and show:

(5) $x1_0 \in \bigcup\{\{a \mid_a a \in A_0 \wedge a \sim x\} \mid_x x \in A_0\}.$

In order to show (5) we have to show

(29) $\exists_{x4} x1_0 \in x4 \wedge x4 \in \{\{a \mid_a a \in A_0 \wedge a \sim x\} \mid_x x \in A_0\}.$

In order to solve (29) we have to find $x4^*$ such that

(30) $x1_0 \in x4^* \wedge \exists_x (x \in A_0 \wedge x4^* = \{a \mid_a a \in A_0 \wedge a \sim x\}).$

Since (6) matches a part of (30) we try to instantiate, i.e. let now $x := x1_0$.
Thus, by (30), we choose $x4^* := \{a \mid_a a \in A_0 \wedge a \sim x1_0\}$.

Now, it suffices to show

(32) $x1_0 \in A_0 \wedge x1_0 \in \{a \mid_a a \in A_0 \wedge a \sim x1_0\}$.

Proof of (32.1) $x1_0 \in A_0$:
Formula (32.1) is true because it is identical to (6).
Proof of (32.2) $x1_0 \in \{a \mid_a a \in A_0 \wedge a \sim x1_0\}$:

In order to prove (32.2) we have to show:

(33) $x1_0 \in A_0 \wedge x1_0 \sim x1_0$.

Formula (33), using (4), is implied by:

(34) $x1_0 \in A_0$.

Formula (34) is true because it is identical to (6). □

We briefly comment on the essential steps in the proof:

- The proof starts with a P-phase, in which the universally quantified implication in the proof goal is reduced by natural deduction inference rules for predicate logic from the special prover `BasicND`, see (1) and (2).
- In a C-phase, the special prover `QR` rewrites the goal and the knowledge base using the definitions in the knowledge base, see (3) and (4).
- The prover switches back again to a P-phase, but now the `STP` prover reduces set equality $X = Y$ to the two subgoals $X \subseteq Y$ and $X \supseteq Y$. In fact, the inference rule for set equality reduces the subgoals by Definition of '$\subset$' immediately, see (5) and (6).
- For proving the first subgoal (6), staying in a P-phase, `STKBR` expands membership in a union and a set quantifier in the knowledge base in two subsequent level saturation runs, see (7), (8), (9) and (10).
- In a C-phase, `QR` uses the equality (10) for rewriting (8) into (23).
- In the final P-phase, expanding membership proves the subgoal (6).
- For proving the second subgoal (5), first `STP` reduces membership in a union during a P-phase into the existential goal (29).
- The set theory prover enters an S-phase. The goal (29) has the special structure $\exists_{x4} x1_0 \in x4 \wedge x4 \in \{T_x \mid_x P_x\}$, which can be handled by rule 'IntroSolveConstant' from Section 7. Thus, the existential quantifier is eliminated by introducing the solve constant $x4^*$, and the expansion of the inner membership $x4^* \in \{T_x \mid_x P_x\}$ introduces another existential quantifier (now for x), see (30).
- The existential sub-formula in (30) is solved for x by unification with formulae in the knowledge base. In fact, in this example *matching* is sufficient, but we provide unification in this step for the general case. Having solved for x, the solve constant $x4^*$ can be instantiated from the equational sub-formula $x4^* = \ldots$ in (30), reducing the solve problem (30) again to a proof problem, see (32).
- In the P-phase, the goal (32) is split using general predicate logic, subgoal (32.1) is trivially true, and subgoal (32.2) is handled first by a set theory specific proof rule from `STP`, see (33).

- Finally, the goal (33) is proved by simple rewriting using implications from the knowledge base in a C-phase, see (34).

8.2.5. An example of "Weak Performance" of the set theory prover

Proof (*SET624*). $\underset{B,C,D}{\forall} B \cap (C \cup D) \neq \{\} \Leftrightarrow B \cap C \neq \{\} \vee B \cap D \neq \{\}$.

For proving (SET624) we take all variables arbitrary but fixed and prove:

(1) $B_0 \cap (C_0 \cup D_0) \neq \{\} \Leftrightarrow B_0 \cap C_0 \neq \{\} \vee B_0 \cap D_0 \neq \{\}$.

Direction from left to right:
We assume

(3) $B_0 \cap (C_0 \cup D_0) \neq \{\}$

and show

(2) $B_0 \cap C_0 \neq \{\} \vee B_0 \cap D_0 \neq \{\}$.

From (3) we know that we can choose an appropriate value such that

(6) $B1_0 \in B_0 \cap (C_0 \cup D_0)$.

From (6) we can infer

(8) $B1_0 \in B_0$,

(9) $B1_0 \in C_0 \cup D_0$.

From (9) we can infer

(11) $B1_0 \in C_0 \vee B1_0 \in D_0$.

We prove (2) by proving the first alternative negating the other(s).
We assume

(13) $\neg(B_0 \cap D_0 \neq \{\})$.

We now show

(12) $B_0 \cap C_0 \neq \{\}$.

Formula (12) means that we have to show that

(14) $\underset{B2}{\exists} B2 \in B_0 \cap C_0$.

We prove (14) by splitting up the intersection into its individual components:
We have to prove:

(15) $\underset{B2}{\exists} B2 \in B_0 \wedge B2 \in C_0$.

Formula (13) is simplified to

(16) $B_0 \cap D_0 = \{\}$.

From (16) we can infer

(17) $\underset{B3}{\forall} B3 \notin B_0 \cap D_0$.

From (17) we can infer

(18) $\underset{B3}{\forall} B3 \notin B_0 \vee B3 \notin D_0$.

We prove (15) by case distinction using (11).
Case (11.1) $B1_0 \in C_0$:
Now, let $B2 := B1_0$. Thus, for proving (15) it is sufficient to prove:

(20) $B1_0 \in B_0 \wedge B1_0 \in C_0$.

Proof of (20.1) $B1_0 \in B_0$:
Formula (20.1) is true because it is identical to (8).
Proof of (20.2) $B1_0 \in C_0$:
Formula (20.2) is true because it is identical to (11.1).
Case (11.2) $B1_0 \in D_0$:
From (8), by (18), we obtain:

(29) $B1_0 \notin D_0$.

Formula (15) is proved because (29) and (11.2) are contradictory.
Direction from right to left:
We assume

(5) $B_0 \cap C_0 \neq \{\} \vee B_0 \cap D_0 \neq \{\}$

and show

(4) $B_0 \cap (C_0 \cup D_0) \neq \{\}$.

Formula (4) means that we have to show that

(30) $\underset{B4}{\exists} B4 \in B_0 \cap (C_0 \cup D_0)$.

We prove (30) by splitting up the intersection into its individual components:
We have to prove:

(31) $\underset{B4}{\exists} B4 \in B_0 \wedge B4 \in C_0 \cup D_0$.

We prove (31) by case distinction using (5).
Case (5.1) $B_0 \cap C_0 \neq \{\}$:
From (5.1) we know that we can choose an appropriate value such that

(32) $B5_0 \in B_0 \cap C_0$.

From (32) we can infer

(34) $B5_0 \in B_0$,

(35) $B5_0 \in C_0$.

Now, let $B4 := B5_0$. Thus, for proving (31) it is sufficient to prove:

(38) $B5_0 \in B_0 \wedge B5_0 \in C_0 \cup D_0$.

We prove the individual conjunctive parts of (38):
Proof of (38.1) $B5_0 \in B_0$:
Formula (38.1) is true because it is identical to (34).
Proof of (38.2) $B5_0 \in C_0 \cup D_0$:
In order to prove (38.2) we may assume

(40) $B5_0 \notin D_0$

and show:

(39) $B5_0 \in C_0$.

(Note, that in all other cases the formula (38.2) trivially holds!)
Formula (39) is true because it is identical to (35).
Case (5.2) $B_0 \cap D_0 \neq \{\}$:
From (5.2) we know that we can choose an appropriate value such that

(41) $B6_0 \in B_0 \cap D_0$.

From (41) we can infer

(43) $B6_0 \in B_0$,

(44) $B6_0 \in D_0$.

Now, let $B4 := B6_0$. Thus, for proving (31) it is sufficient to prove:

(47) $B6_0 \in B_0 \wedge B6_0 \in C_0 \cup D_0$.

We prove the individual conjunctive parts of (47):
Proof of (47.1) $B6_0 \in B_0$:
Formula (47.1) is true because it is identical to (43).
Proof of (47.2) $B6_0 \in C_0 \cup D_0$:
In order to prove (47.2) we may assume

(49) $B6_0 \notin D_0$

and show:

(48) $B6_0 \in C_0$.

(Note, that in all other cases the formula (47.2) trivially holds!)
Formula (48) is proved because (49) and (44) are contradictory. □

Although the prover does not generate any failing branches in this example it is substantially slower than the CASC provers. SPASS shows similar behavior like the *Theorema* prover in that (SET624) is the example in which SPASS performs by far worst. Most probably, the reason for the weak performance of the *Theorema* set theory prover is an inefficient implementation of matching the existentially quantified variable against constants available in the knowledge base, which is needed several times in this example. Note, however, that the proof is straightforward and easy to comprehend for a human reader.

9. Conclusion

This paper describes the design and the implementation of an automated prover for Zermelo–Fraenkel set theory (ZF) in the frame of the *Theorema* system. In particular, we describe how the PCS paradigm for structuring automated theorem provers, which has already been used in other provers provided in *Theorema*, has been accommodated to set theory. The prover is intended to support mathematicians working in arbitrary areas of mathematics that are formulated *using ZF* rather than for proving theorems of ZF from the axioms. This means, we aim at proving theorems in the flavor of the examples shown in Sections 8.2.2 and 8.2.4 much more than most of the examples from the TPTP library. The proofs shown in Section 8 demonstrate that the *Theorema* set theory prover is able to produce proofs of non-trivial theorems in a human-comprehensible style. On average, the computing times for automatically generating the formatted proofs are comparably low. The set theory prover as described in this paper is contained in the public version of the *Theorema* system, which is freely available at http://www.theorema.org.

From the point of view of prover design, the set theory prover is the first prover in the *Theorema* system that interfaces *proving* with *computing* based on available language semantics. The special provers `STKBR` and `STC` will be used as models for future special provers requiring access to the *Theorema* computation engine. Further investigations will be necessary in order to handle conditional rewriting more efficiently and to improve the S-phase by developing more powerful special solvers and by interfacing solvers available in the computer algebra and the constraint solving community.

References

Afshordel, B., Hillenbrand, T., Weidenbach, C., 2001. First order atom definitions extended. In: Nieuwenhuis, R., Voronkov, A. (Eds.), LPAR 2001. In: LNAI, vol. 2250. Springer Verlag, Berlin, Heidelberg, pp. 309–319.

Bernays, P., Fraenkel, A., 1968. Axiomatic set theory. In: Studies in Logic and the Foundations of Mathematics, 2nd ed. North-Holland Publishing Company.

Buchberger, B., 1985. Gröbner bases: an algorithmic method in polynomial ideal theory. In: Bose, N. (Ed.), Multidimensional Systems Theory. D. Reidel Publishing Company, Dordrecht, Boston, Lancaster, pp. 184–232.

Buchberger, B., 1996. Mathematics: an introduction to mathematics integrating the pure and algorithmic aspect. In: Volume I: A Logical Basis for Mathematics. In: Lecture Notes for the Mathematics Course in the First and Second Semester at the Fachhochschule for Software Engineering in Hagenberg, Austria.

Buchberger, B., 1999. Theory Exploration Versus Theorem Proving. In: Armando, A., Jabelean, T. (Eds.), Electronic Notes in Theoretical Computer Science, vol. 23-3. Elsevier, pp. 67–69. CALCULEMUS Workshop, University of Trento, Trento, Italy.

Buchberger, B., 2001. The PCS prover in Theorema. In: Moreno-Diaz, R., Buchberger, B., Freire, J. (Eds.), Proceedings of EUROCAST 2001 (8th International Conference on Computer Aided Systems Theory — Formal Methods and Tools for Computer Science). In: Lecture Notes in Computer Science, vol. 2178, 2201. Springer, Berlin, Heidelberg, New York, pp. 469–478.

Buchberger, B., Vasaru, D., 2000. The Theorema PCS Prover. Jahrestagung der DMV, Dresden, September 18–22.

CADE-18 ATP System Competition (CASC-18), n.d. http://www.cs.miami.edu/~tptp/CASC/18/.

Collins, G. E., 1975. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In: Second GI Conference on Automata Theory and Formal Languages. In: LNCS, vol. 33. Springer Verlag, Berlin, pp. 134–183.

Ebbinghaus, H., 1979. Einführung in die Mengenlehre, 2nd ed., Wissenschaftliche Buchgesellschaft Darmstadt. ISBN 3-534-06709-6.

Formisano, A., 2000. Theory-based resolution and automated set reasoning. Ph.D. Thesis. Universita degli Studi di Roma La Sapienza.

Ganzinger, H., Stuber, J., 2003. Superposition with equivalence reasoning and delayed clause normal form transformation. In: Proc. 19th Int. Conf. on Automated Deduction (CADE-19). In: LNCS, vol. 2741. Miami, USA, pp. 335–349.

Halmos, P., 1960. Naive Set Theory. D. Van Nostrand Company, Princeton, NJ (Reprinted by Springer-Verlag, New York, 1974, ISBN: 0-387-90092-6 (Springer-Verlag edition)).

Kriftner, F., 1998. Theorema: The language. In: Buchberger, B., Jebelean, T. (Eds.), Proceedings of the Second International Theorema Workshop, pp. 39–54. RISC report 98-10.

McCarthy, J., 1964. A tough nut for proof procedures. Stanford AI project memo.

McCarthy, J., 1995. The mutilated checkerboard in set theory. In: Matuszewski, R. (Ed.), The QED Workshop II, Warshaw University, pp. 25–26. Technical Report L/1/95.

Piroi, F., 2004. Tools for using automated provers in mathematical theory exploration. Ph.D. Thesis. RISC Institute, University of Linz.

Quine, W., 1963. Set Theory and its Logic. Belknap Press of Harvard University Press, Cambridge, Massachusetts.

Russell, B., Whitehead, A., 1910. Principia Mathematica. Cambridge University Press (Reprinted 1980).

Shoenfield, J.R., 1967. Mathematical Logic. Logic, Addison Wesley Publishing Company.

Tomuta, E., 1998. An architecture for combining provers and its applications in the Theorema system. Ph.D. Thesis. The Research Institute for Symbolic Computation, Johannes Kepler University. RISC report 98-14.

TPTP: Thousands of Problems for Theorem Provers, n.d. http://www.cs.miami.edu/~tptp/.

Vasaru-Dupré, D., 2000. Automated theorem proving by integrating proving, solving and computing. Ph.D. Thesis. RISC Institute. RISC report 00-19.

Windsteiger, W., 2001a. A set theory prover in *Theorema*: implementation and practical applications. Ph.D. Thesis. RISC Institute, http://www.risc.uni-linz.ac.at/people/wwindste/publications.html.

Windsteiger, W., 2001b. On a solution of the mutilated checkerboard problem using the *Theorema* set theory prover. In: Linton, S., Sebastiani, R. (Eds.). Proceedings of the Calculemus 2001 Symposium, http://www.risc.uni-linz.ac.at/people/wwindste/publications.html.

Available online at www.sciencedirect.com
ScienceDirect
Journal of Applied Logic 4 (2006) 470–504
JOURNAL OF APPLIED LOGIC
www.elsevier.com/locate/jal

Theorema: Towards computer-aided mathematical theory exploration

Bruno Buchberger [a], Adrian Crăciun [a], Tudor Jebelean [a], Laura Kovács [a], Temur Kutsia [a,*], Koji Nakagawa [a], Florina Piroi [b], Nikolaj Popov [a], Judit Robu [a], Markus Rosenkranz [b], Wolfgang Windsteiger [a]

[a] *Research Institute for Symbolic Computation, Johannes Kepler University, Altenbergerstraße 69, A-4040 Linz, Austria*
[b] *Johann Radon Institute for Computational and Applied Mathematics, Austrian Academy of Sciences, Altenbergerstraße 69, A-4040 Linz, Austria*

Available online 17 November 2005

Abstract

Theorema is a project that aims at supporting the entire process of mathematical theory exploration within one coherent logic and software system. This survey paper illustrates the style of *Theorema*-supported mathematical theory exploration by a case study (the automated synthesis of an algorithm for the construction of Gröbner Bases) and gives an overview on some reasoners and organizational tools for theory exploration developed in the *Theorema* project.

Keywords: Mathematical assistant; Automated reasoning; Theory exploration; "Lazy Thinking"; Theorema

* Corresponding author.
E-mail address: temur.kutsia@risc.uni-linz.ac.at (T. Kutsia).

1570-8683/$ – see front matter
doi:10.1016/j.jal.2005.10.006

1. Introduction

1.1. Aims of Theorema*: A brief overview*

Theorema is a project and a software system that aims at supporting the entire process of mathematical theory exploration: invention of mathematical concepts, invention and verification (proof) of propositions about concepts, invention of problems formulated in terms of concepts, invention and verification (proof of correctness) of algorithms solving problems, and storage and retrieval of the formulae invented and verified during this process. This integral objective was already formulated at the very beginning of the *Theorema* project; see, e.g., [9]. In particular, we emphasize

- a holistic view of the mathematical theory exploration process [11] as opposed to proving individual, isolated theorems;
- proof presentation in a "natural", mathematical textbook style that should make it easy for human readers to understand and check the proofs generated automatically;
- the presentation of logic formulae in a natural two-dimensional syntax easily changeable by the user without changing the internal abstract syntax;
- the view and usage of higher-order equational logic as a programming language internal to predicate logic, which makes it possible to execute verified algorithms within the same system in which the verification was done;
- automated proof generation as opposed to automated proof checking;
- efficient proof generation in special theories—like geometry, analysis, combinatorics—using algebraic algorithms as black box inference rules; (this links past research expertise of the *Theorema* group, notably in the area of Gröbner Bases theory [7,8], to the current logic-oriented research goals);
- automatically proving the algebraic methods that are later used as a part of special theory inferencing;
- the user-controlled linkage of mathematical knowledge bases to the logic system;
- the usage of an advanced front end (including publishing, graphics, and web-tools) of a mathematical software system (namely, Mathematica [93]).

In the *Theorema* project we developed methods, system components and tools that cover parts of the entire research plan. In particular, we have

- A basic implementation frame: a symbolic computation software system Mathematica. Note that we do not rely on the mathematical algorithms library of such a system but only on the programming language frame. The user can call the Mathematica algorithms in *Theorema* in a controlled way.
- A mathematical language as a common frame for both nonalgorithmic and algorithmic mathematics. Basically, it is a higher order logic language extended with sequence variables (i.e., variables that can be instantiated with finite, possibly empty sequences of terms; see [53]). The interpreter of the algorithmic part of the language that consists of "executable formulae" (function definitions using induction and bounded quantifiers) is readily available within Mathematica. The semantics of the algorithmic part

consists of computation rules and basic operations on numbers, sets, and tuples. For arithmetic operations on natural, integer, and rational numbers the *Theorema* semantics may access the arithmetic rules from Mathematica, if told to.
- Various reasoners: "internal" ones, implemented within *Theorema*, and "external" ones, linked to the system. All the "internal" reasoners follow a common design: They are composed of individual rules applicable to certain reasoning situations (goals and knowledge bases). The rules are grouped into special modules that can be combined into a reasoner using various strategies. The actual generation of the output is guided by the *common search procedure*. The output is represented by a *global reasoning object* that follows a common structure in order to allow a homogeneous display of the output independent of which reasoner generated it. Note that a particular reasoner need not understand the whole *Theorema* syntax.
- A general facility that allows the presentation of reasoner outputs in natural language. All the *Theorema* "internal" reasoners produce output that imitate "natural" reasoning styles of human mathematicians.
- Mechanisms for the automatic generation of complicated knowledge bases from the algebraic properties of given domains and the definition of functors.

For a more detailed account and the bibliography roadmap on these issues we refer to the survey papers [19,20].

The current paper is another survey that concentrates on methods and tools for theory exploration in *Theorema* which have been developed in the last four years and, hence, are not contained in [19,20]. Here we only give an overview and give references to the articles and reports where these methods and tools are described in detail.

1.2. Methods and tools for theory exploration in Theorema

An example of a theory exploration method is Lazy Thinking [13–15] that relies on algorithm schemata and the automated analysis of failing correctness proofs. It is used in algorithm synthesis—a stage in theory exploration when one tries from a given knowledge base and algorithm schemata to derive an algorithm that fulfils a given specification. The method turns out to be powerful enough to synthesize not only toy examples like sorting algorithms [28] but also nontrivial algorithms like the usual algorithm for computing Gröbner Bases of polynomial ideals [8,15].

A user (working mathematician) who intends to explore a theory using *Theorema* interacts with three blocks of system components: reasoners, organizational tools, and knowledge bases; see Fig. 1. For instance, she may construct theories and add them to the knowledge bases with the management tools; invent new concepts, propositions, problems, and algorithms using the schema libraries; prove propositions and verify algorithms with the reasoners; display proofs using the presentation tools, etc.

The core part of mathematical theory exploration is reasoning: proving, computing, and solving. All these activities are done either in the general frame of "pure" (higher-order) predicate logic or in various special theories specified by suitable axioms (in the same logic). For example, resolution is a universal proving method, β-reduction is a universal computing engine, and syntactic unification can be understood as a solver with no

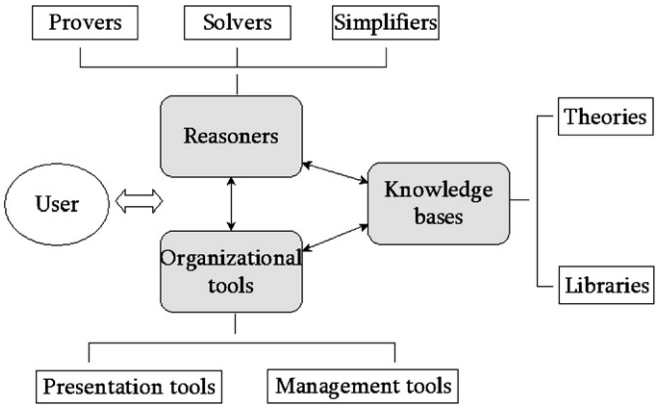


Fig. 1. *Theorema* components for theory exploration.

special knowledge. The reasoning activities can be "custom-tailored" to particular theories. For example: in the domain of naturals, any induction prover can be understood as a special prover; a canonical reduction system induced by a given equational theory provides a mechanism of computation in the given theory; Collins's algorithm for cylindrical algebraic decomposition (CAD) is a special solver over the theory of real closed fields. *Theorema* aims at providing a uniform (logic and software) frame for these activities. Reasoners are accessed by the call

$$\mathit{Action}[\mathit{entity},\ \text{using} \rightarrow \mathit{knowledge\text{-}base},\ \text{by} \rightarrow \mathit{reasoner},\ \mathit{options}],$$

where *Action* is the desired action that the reasoner should perform, i.e., Prove, Compute, or Solve; *entity* is the mathematical entity, to which the action should apply, e.g., a proposition in the case of proving or just an expression in the case of computing; *knowledge-base* is the knowledge base with respect to which the action should be performed; *reasoner* is the concrete reasoner that should perform the action; *options* are possible options to be given to the reasoner in order to influence its behavior.

Currently *Theorema* contains (or is linked with) about 30 (automatic or semi-automatic) reasoners. We describe in this paper only the recent developments: the basic reasoner, the equational prover, proving by S-decomposition, the special solver and simplifier for the theory of differential equations, the geometry prover, the verification condition generators for imperative and functional programs, and the interface to external systems. The other *Theorema* reasoners that have been implemented earlier, documented in [19,20], include: the prover for classical predicate logic that implements a sequent calculus with metavariables; the PCS (Prove–Compute–Solve) prover that extends the predicate logic prover with a special method that, essentially, reduces proving to solving (the method was successfully used in proving problems in elementary analysis where proving was reduced to solving real constraints by Collins's CAD algorithm [26]); the prover for Zermelo–Fraenkel set theory that extends the PCS prover by special inference rules for reasoning with sets; induction provers for natural numbers, lists, and sequence variables; the Gröbner Bases prover for boolean combinations involving polynomial equalities and inequalities; the Gosper–

Zeilberger prover for problems involving combinatorial identities over integers, and other reasoners. We do not discuss them in this paper.

Traditional automated provers have no integrated mathematical knowledge. It makes it difficult to use them in mathematical problem solving. A system that assists mathematicians must have access to mathematical knowledge. In *Theorema* the user can build mathematical domains using functors, define and manipulate theories, access external knowledge bases, and build and use concept, theorem, problem and algorithm schema libraries.

A special remark should be made about using types and sorts in *Theorema*. The language of *Theorema* is untyped. Therefore, if a type or sort information is needed, it is in general handled by unary predicates or sets (in case one decides to work within a set theory). However, particular reasoners can implement rules to deal with such an information in a special way.

The user-friendliness of a mathematical assistant is important for its acceptance and performance. *Theorema* organizational tools are designed for this purpose. This concerns not only the graphical-user interface, but also components that help users in managing (accessing, updating, browsing) mathematical knowledge bases, developing and presenting proofs in a human-oriented way, using traditional mathematical notation, and extending syntax. In this paper we describe three such tools: focus windows for displaying mathematical proofs, label management for organizing knowledge bases, and logicographic symbols as a powerful extension of conventional mathematical syntax.

The paper is organized as follows: First, in Section 2 we introduce Lazy Thinking as a particular *method* of theory exploration, which combines reasoners and organizational tools in a certain way and show its power on a synthesis of the usual algorithm for constructing Gröbner Bases. Next, we describe various new *tools* for theory exploration available in *Theorema*: reasoners in Section 3 and organizational tools in Section 4. Related work is discussed in Section 5 and conclusions are given in Section 6.

The *Theorema* system and publications are available to download from the project web page: http://www.theorema.org/.

2. The Lazy Thinking method

2.1. General idea

We recently proposed in [14] a model for theory exploration based on schemata that represent condensed mathematical knowledge of various types (definitions, propositions, problems, algorithms). In this model, given a *theory exploration situation*, that is,

- a knowledge base **K** (a structured collection of logic formulae that describe notions—predicates and functions—and their properties),
- and a library of schemata **L** (conceptually, higher-order formulae),

one *step of theory exploration* expands the knowledge base by

- inventing new notions (by using definition schemata);

– inventing (by proposition schemata) and proving or disproving (using the available proving mechanisms) propositions about the new notions;
– inventing problems (by problem schemata) that involve the notions;
– inventing and verifying methods (algorithms) to solve the problems.

As a particular contribution to the invention of methods (algorithms) that solve problems based on algorithm schemata we introduced *the method of Lazy Thinking*. (Here we can only summarize the main ideas of the method, all details are given in [13–15].) The method proceeds as follows: We start from

– an exploration situation, i.e., a knowledge base **K** and a library of algorithm schemata **L** (formulae that define algorithms in terms of auxiliary unknown subalgorithms, together with an appropriate inductive proof strategy), and
– a problem **P**, i.e., a formula of the form $\forall_x\, Q[x, A[x]]$, where $Q[x, y]$ describes the relation between input x and output y, and A is the algorithm to be synthesized. (Q can be any predicate logic formula. In the example in the next subsection Q is defined by

$$\underset{F,G}{\forall}\, Q[F, G] \Leftrightarrow \text{is-finite}[G] \wedge \text{is-Gröbner-basis}[G] \wedge \text{ideal}[F] = \text{ideal}[G].)$$

The algorithm A that fulfills the specification **P** is determined as follows:

(1) Select a new schema from **L**, add it to the knowledge base and try to prove the correctness theorem, using the proof strategy indicated by the algorithm schema. The proof is likely to fail because the properties of the unknown auxiliary algorithms introduced by the algorithm schema are yet unknown.
(2) A specification generation algorithm described in [13] analyzes the failing proof and generates specifications of the unknown auxiliary algorithms that allow the proof to get over the failing point. This specification generation algorithm is an essential part of the method. It identifies, analyzes and generalizes temporary assumptions and goals in a specific way.
(3) Add the specifications to the knowledge base and repeat the previous step until the proof succeeds. If the proof does not succeed, go back to step (1).
(4) Once the proof is completed, the result of the Lazy Thinking method is the proof that the algorithm A, as defined by the algorithm schema, fulfills the specification **P** provided that the auxiliary algorithms introduced by the schema meet the specifications generated.

In order to complete the synthesis process, we have to find the auxiliary algorithms. We have two possibilities: Either appropriate algorithms are already available in the knowledge base, or we have to apply the method of Lazy Thinking again for synthesizing the auxiliary algorithms. Of course, there is no guarantee that the recursive application of the Lazy Thinking will always terminate, i.e., Lazy Thinking is not an algorithm. However, it terminates on many interesting examples. One such example is described below.

2.2. *Example: Synthesizing an algorithm for Gröbner bases by Lazy Thinking*

The Lazy Thinking method is powerful enough to deal with the synthesis of nontrivial algorithms, such as an algorithm for the construction of Gröbner Bases [7,8] as we have shown in [15]. Below we give the input and output for this case study in order to illustrate the style in which *Theorema* supports the mathematical exploration process. The full implementation of this case study has still to overcome a couple of technical problems.

The problem consists in finding, automatically, an algorithm GB that satisfies the specification

$$\textbf{Problem}[\text{“Gröbner Bases”},\ \underset{F}{\forall} \wedge \begin{cases} \text{is-finite}[\text{GB}[F]] \\ \text{is-Gröbner-basis}[\text{GB}[F]]\,] \\ \text{ideal}[F] = \text{ideal}[\text{GB}[F]] \end{cases}.$$

(In order not to distract from the main flow of the exploration, we omit here all type specifications, e.g., that F should range over finite sets of multivariate polynomials.) Of course, we have to know a lot about the ingredient auxiliary notions like “is-Gröbner-basis”, “is-finite”, etc. This knowledge can be compiled, for example, by the construct

$$\textbf{Theory}[\text{“Gröbner Bases prerequisites”},$$
$$\underset{G}{\forall}\ \text{is-Gröbner-basis}[G] \ \Leftrightarrow\ \text{is-confluent}[\rightarrow_G]$$
$$\underset{G,h_1,h_2}{\forall}\ h_1 \rightarrow_G h_2 \ \Leftrightarrow\ \underset{g\in G}{\exists} \wedge \begin{cases} \text{lp}[g] \mid \text{lp}[h_1] \\ h_2 = h_1 - (\text{lm}[h_1]/\text{lm}[g])g \end{cases}]$$
$$\cdots$$

In fact, this theory can be structured hierarchically by grouping theories within theories, e.g., the theory of polynomial rings, reduction theory and ideal theory.

The Lazy Thinking method proceeds now by trying out algorithm schemata (taken from a library of algorithm schemata): An algorithm schema that is appropriate for this particular synthesis problem is the so-called “critical pair/completion” schema, that describes the *unknown* algorithm GB in terms of *unknown* auxiliary algorithms *lc* and *df*:

$$\text{GB}[F] = \text{GB}[F, \text{pairs}[F]]$$
$$\text{GB}[F, \langle\,\rangle] = F$$
$$\text{GB}[F, \langle\langle g_1, g_2\rangle, \bar{p}\rangle] =$$
$$\text{where}[f = lc[g_1, g_2], h_1 = \text{trd}[\text{rd}[f, g_1], F], h_2 = \text{trd}[\text{rd}[f, g_2], F],$$
$$\begin{cases} \text{GB}[F, \langle\bar{p}\rangle] & \Leftarrow h_1 = h_2 \\ \text{GB}[F \frown df[h_1, h_2],\ \langle\bar{p}\rangle \asymp \langle\langle F_k, df[h_1, h_2]\rangle \underset{k=1,\ldots,|F|}{\mid} \rangle] & \Leftarrow \text{otherwise}\,]. \end{cases}$$

In the schema a couple of *known* algorithms appear, like “pairs” (forming all pairs of objects in F), “rd” (one reduction step), “trd” (total reduction), “$\frown$” (append), “$\asymp$” (concatenate), etc. Note that we use the sequence variable $\bar{p}$, for which any finite number of

terms can be substituted. In fact, the "critical pair/completion" schema is a quite general one that incorporates, for instance, Knuth–Bendix type or resolution type procedures.

Here, "trying out" means to start an (automated) proof of the correctness of GB as a candidate for the unknown algorithm to be synthesized. In our case the *Theorema* prover suitable for this task is a relatively simple rewrite prover (since the necessary induction is already contained in Newman's lemma). This proof will, of course, fail because nothing is known about *lc* and *df*. The core of the method is an algorithm that analyzes the failing proof, and automatically generates conditions on *lc* and *df* under which the correctness proof will succeed. These conditions can now be viewed as specifications for the unknown auxiliary algorithms *lc* and *df*: If we manage to find algorithms satisfying these specifications then the above algorithm schema becomes an algorithm that satisfies the initial specification, i.e., constructs a Gröbner basis for any input F. Note that, along with the synthesis of the algorithm, the system also provides a proof of its correctness. In the given example, the automatically generated specifications of *lc* and *df* are

$$\underset{g_1,\,g_2,\,p}{\forall} \wedge \begin{cases} \mathrm{lp}[g_1] \mid lc[g_1, g_2] \\ \mathrm{lp}[g_2] \mid lc[g_1, g_2] \\ \wedge \begin{cases} \mathrm{lp}[g_1] \mid p \\ \mathrm{lp}[g_2] \mid p \end{cases} \Rightarrow (lc[g_1, g_2] \mid p) \end{cases} \quad \text{and} \quad \underset{h_1,\,h_2}{\forall} h_1 \downarrow^*_{df[h_1,h_2]} h_2.$$

It is now very easy (and can again be done automatically by Lazy Thinking, using the available knowledge on the theory of polynomials) to find appropriate algorithms "lc" and "df" that satisfy these specifications. Namely,

$$\mathrm{lc}[g_1, g_2] = \text{least-common-multiple}[lp[g_1], lp[g_2]],$$
$$\mathrm{df}[h_1, h_2] = h_1 - h_2.$$

The algorithm GB together with the algorithms for "lc" and "df" is now executable within *Theorema*. Note that the algorithm "lc" synthesized automatically by the Lazy Thinking method constitutes the essential part of the algorithmic Gröbner Bases theory. It has not been synthesized so far by any other method or system and, hence, constitutes a major example of the theory exploration potential of *Theorema*.

3. Reasoners

In this section we describe reasoners developed recently in the *Theorema* project. We start first with general purpose reasoners (the basic reasoner, the S-decomposition method, and the equational prover), then describe some special ones (the solver and simplifier for a special theory of differential equations, the geometry prover, and the reasoners for program verification), and finally show how *Theorema* can interface external reasoning systems. All these reasoners, together with the other provers, solvers, and simplifiers of the *Theorema* system, can be combined under the user control in various ways for theory exploration and applied either completely automatically, or interactively using a special mechanism that guides the reasoning process. This mechanism is a handy tool since most of the mathematical theorems are hard to prove completely automatically. Using it, one can choose

between fully automatic and interactive, stepwise proof development, can easily navigate through the proof object, can inspect proof situations, can provide various hints to the prover (e.g., suitable instantiations), can add formulae to and remove formulae from the temporary knowledge base of the proof, can choose a different reasoner, add or remove branches in the proof, etc. Details of this mechanism are described in [67].

3.1. The Theorema *Basic Reasoner*

The *Theorema* Basic Reasoner combines special features already available in the PCS prover and the set theory prover [92]. From the PCS prover it uses the standard inference rules for first-order predicate logic and the rules that use quantified equalities, equivalences, and implications in the knowledge base for rewriting. In addition, the reasoner is extended with special inference rules for language constructs like the "such-that"—or the "the-unique"—quantifier. From the set theory prover it uses the interface for incorporating computations into proofs. The interface applies the *Theorema* computation engine for the algorithmic fragment of the *Theorema* language in order to simplify parts of formulae. The resulting Basic Reasoner is a general purpose prover that understands almost the entire language available in the *Theorema* syntax. With its access to computation facilities we find it to be appropriate for the type of undergraduate proving exercises that often rely on simplification by arithmetic computations combined with predicate logic reasoning. To illustrate this point of view, we describe the key steps of the proof of the irrationality of $\sqrt{2}$. We assume the positive reals as the universe and we want to prove

Proposition["$\sqrt{2}$ irrational", $\neg\text{rat}[\sqrt{2}]$]

using the knowledge about positive real numbers

Definition["rational", any[r], rat[r]: $\Leftrightarrow \underset{\text{nat}[a,b]}{\exists} (r = \frac{a}{b} \wedge \text{coprime}[a, b])$].

Definition["sqrt", any[x, y], $\sqrt{x} = y \Leftrightarrow y^2 = x$].

Lemma["coprime", any[a, b] with[nat[a] $\wedge$ nat[b]],

$2b^2 = a^2 \Rightarrow \neg\text{coprime}[a, b]$]

by the *Theorema* Basic Reasoner. This is done by executing the command:

Prove[Proposition["sqrt2"], using → ⟨Lemma["coprime"], ...⟩,
built-in → Built-in["Rational Numbers"],
by → BasicReasoner, ProverOptions → {SimplifyFormula → True,
RWCombine → True}].

We remark that nat[a, b] above abbreviates nat[a] $\wedge$ nat[b]. Also, in order the implicit definition of square root to be consistent, it is assumed that $\underset{x}{\forall}\underset{y}{\exists!}\, y^2 = x$ holds over the positive real numbers.

The first steps in the proof transform formulae in the knowledge base. The definition "rational" is expanded and the result is simplified using built-in knowledge available in the semantics of the *Theorema* language. Further, the negated goal, rat[$\sqrt{2}$], is assumed. After several other basic predicate logic reasoning steps we arrive at the following assumption

(6) $\text{coprime}[a_0, b_0] \wedge \text{nat}[a_0] \wedge \text{nat}[b_0] \wedge \sqrt{2} = \frac{a_0}{b_0}$

for arbitrary but fixed a_0 and b_0. Now, nat[a_0] and nat[b_0] are used to instantiate Lemma "coprime", and $\sqrt{2} = \frac{a_0}{b_0}$ is simplified using built-in knowledge from the *Theorema* language semantics. In the example, the option built-in→Built-in["Rational Numbers"] allows the prover to explicitly use built-in rules for operations on rational numbers that rely on Mathematica algorithms. In addition, on explicit user request, the *Theorema* Basic Reasoner is allowed to access special simplification algorithms from Mathematica for performing computational simplification. Specifying the prover option SimplifyFormula→True (default value is False) tells the prover to postprocess any formula obtained from a computation by Mathematica's `FullSimplify` function. `FullSimplify` is a black box simplifier for Mathematica expressions, which uses powerful simplification rules, in particular for arithmetic expressions.

To continue our example proof, using the definition of square root, the simplification of $\sqrt{2} = \frac{a_0}{b_0}$ results in

(9) $2 * {b_0}^2 = {a_0}^2$,

from which we can infer, by an instantiated version of Lemma "coprime",

(10) $\neg\text{coprime}[a_0, b_0]$,

which contradicts the first conjunct in formula (6).

To complete the proof, the Basic Reasoner can, again, be used to prove the auxiliary Lemma "coprime". This lemma can be proved in the universe of natural numbers, which is reflected in the Prove-call by using Built-in["Natural Numbers"] instead of rational numbers.

3.2. Equational prover

Many problems that arise during theory exploration process have an equational form. Equational reasoning belongs to a very long tradition in mathematics and plays an important role in formalizing maths. Here we describe one of the tools *Theorema* provides for equational reasoning: the general equational prover.

The equational prover of *Theorema* [51] is designed for unit equality problems in first-order or higher-order form. A (restricted) usage of Mathematica built-in functions is allowed, if the user explicitly requires it. The input problem can contain sequence variables that are used together with flexible arity symbols and make the language more expressive and flexible. For instance, with sequence variables the idempotence and flatness properties of a flexible arity function f can be expressed in a very concise way:

$\underset{\bar{x},\bar{y},\bar{z},u}{\forall} f[\bar{x}, u, \bar{y}, u, \bar{z}] = f[\bar{x}, u, \bar{y}, \bar{z}]$ for idempotence and $\underset{\bar{x},\bar{y},\bar{z}}{\forall} f[\bar{x}, f[\bar{y}], \bar{z}] = f[\bar{x}, \bar{y}, \bar{z}]$ for flatness. (The overbarred letters are sequence variables.)

The prover has two proving modes: completion and simplification (rewriting/narrowing). The completion proving mode is based on the unfailing completion procedure [2]. The input in the higher-order form is first transformed into the first-order form using Warren's translation method [89]. Then the proving procedure runs on the translated problem and finally the output is translated back into the higher-order form. The user sees only the higher-order input and output.

Mathematica built-in functions can be used in the proving task in the following way: First, the user should state explicitly if she wants a certain function in the proving problem to be interpreted as some Mathematica built-in function. (It is not enough the function in the problem to coincide with a Mathematica function syntactically.) Moreover, such an interpretation is used only for function occurrences in the goal, not in the assumptions. After normalization, the goal is checked for joinability modulo its functions built-in meaning, but the built-ins are not used to derive new goals. After a built-in function is identified, it is trusted and the result of computation is not checked. In this case, the corresponding warning is issued.

We extended the unfailing completion allowing flexible arity functions and sequence variables in equalities. Such problems arise, for example, in the exploration of the theory of tuples [17]. The main difficulty in reasoning with sequence variables is the infinitary unification [52]. However, under certain restrictions it can be made finitary, or even unitary. The equational prover deals exactly with such cases. The unfailing completion is extended for equalities where sequence variables occur only in the last argument positions in the subterms. This restriction, still covering quite a wide range of interesting cases, makes unification unitary. In the simplification mode this restriction is lifted but existential goals for problems with sequence variables are not allowed. In this case matching with sequence variables is sufficient. It is finitary.

Proofs are described by the Proof Communication Language PCL [31]. They are structured into lemmata/propositions. Proofs of universally closed theorems are displayed as equational chains, while those of existential theorems represent sequences of equations. In failing proofs, on the one hand, the theorems which have been proved during completion are given. On the other hand, failed propositions whose proving would lead to proving the original goal are displayed, if there are any. They are obtained from descendants of the goal and certain combinations of their left- and right-hand sides.

To summarize, the strengths of the prover are: the ability to handle sequence variables and problems in the higher-order form, to interface with Mathematica functions, and to generate proofs in a human-oriented style.

3.3. The proof method by S-decomposition

Numerous interesting mathematical notions are defined by formulae that contain a sequence of "alternating quantifiers", i.e., the definitions have the structure $p[x, y] \Leftrightarrow \underset{a}{\forall}\underset{b}{\exists}\underset{c}{\forall}\ldots q[x, y, a, b, c]$. Many notions introduced, for example, in elementary analysis text books (limit, continuity, function growth order, etc.) fall into this class. Therefore, it is

highly desirable that mathematical assistant systems support the exploration of theories about such notions. It is not an easy task: The automation of so-called "epsilon-delta" proofs, typical for the propositions in analysis about notions defined using alternating quantifiers, was since long time considered a practically important challenge for traditional provers; see, e.g., [5,61].

The S-decomposition method is particularly suitable both for proving theorems (when the auxiliary knowledge is rich enough) as well as conjecturing propositions (similar to Lazy Thinking) during the exploration of theories about notions with alternating quantifiers. It can be seen as a further refinement of the Prove-Compute-Solve method implemented in the *Theorema* PCS prover. Essentially, the S-decomposition method is a certain strategy for decomposing the proof into simpler subproofs, based on the structure of the main definition involved. The method proceeds recursively on a group of assumptions together with the quantified goal, until the quantifiers are eliminated, and produces some auxiliary lemmata as subgoals.

We present the method using an example from elementary analysis: limit of a sum of sequences; see [40] for a detailed description of the method. The definition of "f converges to a" is:

$$(\to:) \quad f \to a \;\Leftrightarrow\; \underset{\epsilon}{\forall}\big(\epsilon > 0 \Rightarrow \underset{m}{\exists}\,\underset{n}{\forall}\big(n \geqslant m \Rightarrow \big|f[n] - a\big| < \epsilon\big)\big).$$

For brevity, the type information is not included.

The proof tree is presented in Figs. 2 and 3. Boxes represent proof situations (with the goal on top), unboxed formulae represent auxiliary subgoals, and boxes with double sidebars represent substitutions for the metavariables. The nodes of the proof tree are labeled in the order they are produced.

The first inference expands the definition of "limit", generating the proof situation (2). *S-decomposition is designed for proof situations in which the goal and the main assumptions have exactly the same structure*. In the example they differ only in the instantiations of f and a. *S-decomposition proceeds by modifying these formulae together, such that the similarity of the structure is preserved, until all the quantifiers and logical connectives are eliminated*. The method is specified as a collection of four transformation rules (inferences) for proof situations and a rule for composing auxiliary lemmata. The transformation rules are described below together with their concrete application to this particular proof.

The inference that transforms (2) to (3) eliminates the universal quantifier and has the general formulation below. (Here, for simplicity, we formulate the inferences for two assumptions only, but extending them to use an arbitrary number of assumptions is straightforward.)

$$\underset{x}{\forall}\, P_1[x], \underset{x}{\forall}\, P_2[x] \;\vdash\; \underset{x}{\forall}\, P_0[x] \longmapsto P_1[x_1^*], P_2[x_2^*] \;\vdash\; P_0[x_0]. \tag{$\forall$}$$

Like the existential rule, specified later in this section, this rule combines the well-known techniques for introducing Skolem constants and metavariables. However, S-decomposition comes with a *strategy* of applying them in a certain order. The Skolem constant x_0 is introduced before the metavariables (names for yet unknown terms) x_1^*, x_2^*. In the example we use a simplified version of this rule in which the metavariables do not differ. For other examples (e.g., quotient of sequences) this will not work.

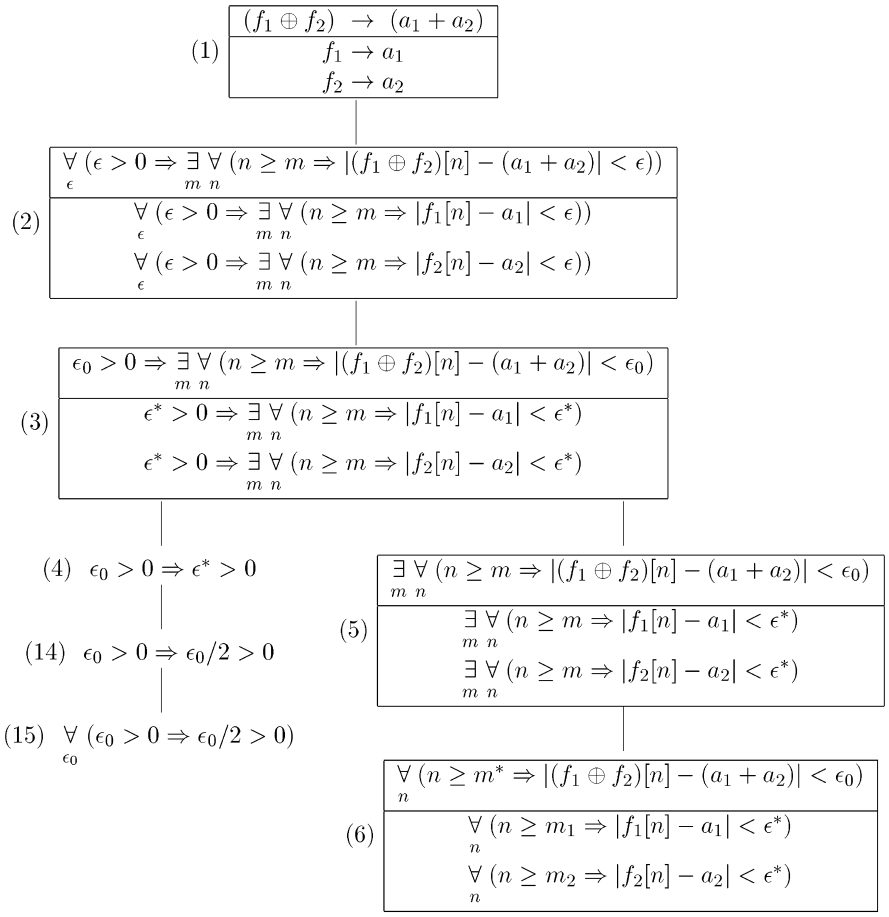


Fig. 2. S-decomposition: First part of the proof tree.

The inference from (3) to (4) and (5) eliminates the implication, and has the general formulation:

$$Q_1 \Rightarrow P_1,\ Q_2 \Rightarrow P_2\ \vdash\ Q_0 \Rightarrow P_0 \longmapsto \begin{cases} Q_0 \Rightarrow Q_1 \wedge Q_2 \\ P_1, P_2\ \vdash\ P_0 \end{cases} \qquad (\Rightarrow)$$

In contrast to the previous rule, this one is not an equivalence transformation (the proof of the right-hand side might fail even if the left-hand side is provable). This rule is applied in the situations when Q_k's are the "conditions" associated with a universal quantifier (as in the example). The formula $Q_0 \Rightarrow Q_1 \wedge Q_2$ is a candidate for an auxiliary lemma, as is formula (4).

The proof proceeds further with the transformation (5)–(6) (formula (14) will be produced later in the proof) given by the following rule:

$$\underset{x}{\exists} P_1[x], \underset{x}{\exists} P_2[x]\ \vdash\ \underset{x}{\exists} P_0[x] \mapsto P_1[x_1], P_2[x_2]\ \vdash\ P_0[x^*] \qquad (\exists)$$

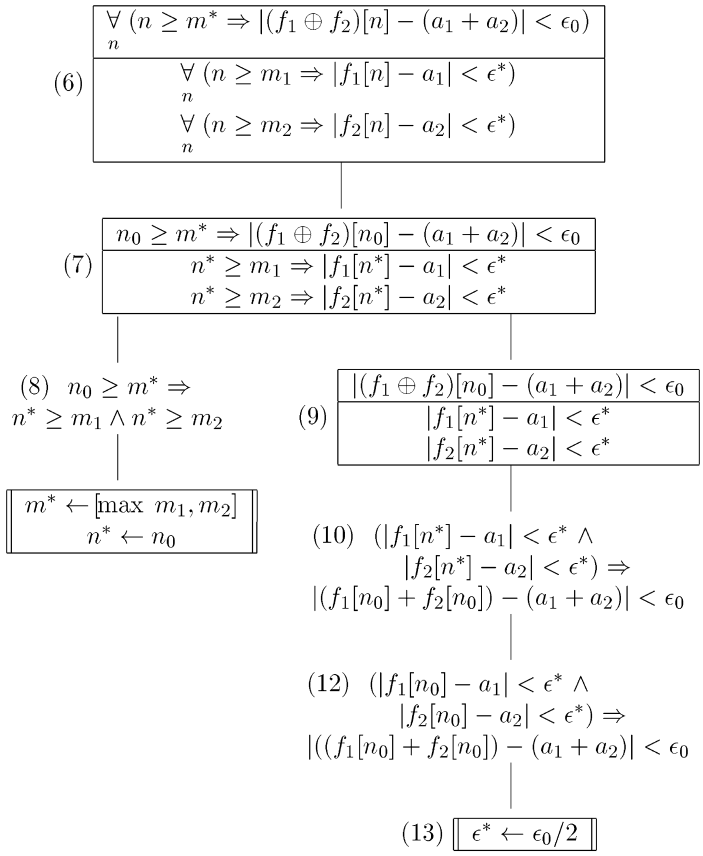


Fig. 3. S-decomposition: Second part of the proof tree.

where x_1 and x_2 are Skolem constants introduced before the metavariable x^*.

Usually, existential quantifiers are associated with conditions upon the quantified variables. In such a case one would obtain conjunctions (analogous to the situation in formula (3), where one obtains implications). The rule for decomposing conjunctions is:

$$Q_1 \wedge P_1,\ Q_2 \wedge P_2\ \vdash\ Q_0 \wedge P_0 \longmapsto \begin{cases} Q_1 \wedge Q_2 \Rightarrow Q_0 \\ P_1, P_2\ \vdash\ P_0. \end{cases} \qquad (\wedge)$$

Similarly to the rule (⇒), this rule produces an auxiliary lemma as a "side effect", using the Q_k's which are, typically, the conditions associated with an existential quantifier. In fact, in the implementation of the method, the rules (∃), (∧) are applied in one step, as are also the rules (∀), (⇒).

However, in this example there is no condition associated to the existential quantifier, therefore this rule is not used.

The proof proceeds by applying rule (∀) to (6), and then the rule (⇒) to (7). Note that the transformation rules proceed from the assumptions towards the goal for existential formulae, and the other way around for universal formulae. If one would illustrate this

process by drawing a line on the formulae in proof situation (2), one obtains an S-shaped curve—thus the name of the method.

Finally, S-decomposition transforms a proof situation having no quantifiers into an implication, thus (9) is transformed into (10), and this finishes the application of S-decomposition to this example. In this moment the original proof situation is decomposed into the formulae (4), (8), and (10). (Obtaining (10) needs an additional inference step, not shown in the figure, which consists in expanding the subterm $(f_1 \oplus f_2)[n_0]$ by the definition of $\oplus$.)

The continuation of the proof is outside the scope of the S-decomposition method. For completing the proof, one needs to find appropriate substitutions for the metavariables, such that the Skolem constants used in each binding are introduced earlier than the corresponding metavariable. For the sake of completeness, we give here a possible follow up (produced automatically by *Theorema*): We assume that the formulae

$$(21)\quad \underset{k,i,j}{\forall}\ (k \geqslant \max[i, j] \Rightarrow k \geqslant i \wedge k \geqslant j),$$

$$(22)\quad \underset{x,y,a,b,\epsilon}{\forall}\ (|x-a| < \tfrac{\epsilon}{2} \wedge |y-b| < \tfrac{\epsilon}{2} \Rightarrow |(x+y)-(a+b)| < \epsilon)$$

are present in the available knowledge as auxiliary assumptions. The prover first tries to "solve" (8), and by matching against (21) obtains the substitution (11). This substitution is applied to (10) producing (12), and by matching the latter against (22), the prover obtains the substitution (13). The substitutions are then applied to the formula (4), which is then generalized (by universal quantification of the Skolem constants) into (15). The latter is presented to the user as suggestions for auxiliary lemmata needed for completing the proof. Of course this subgoal would be also solved if the appropriate assumption was available, however the situation described above demonstrates that the method is also useful for generating conjectures.

The reader may notice that the process of guessing the right order in which the subgoals (4), (8), and (10) should be solved is nondeterministic and may involve some backtracking. This search is implemented in *Theorema* using the principles described in [47].

The auxiliary lemmata can either be proved by domain-specific provers (e.g., CAD within the PCS prover) or can be retrieved from a mathematical knowledge base.

3.4. Solver and simplifier for a special theory of differential equations

The tools we considered so far can be classified as general purpose reasoners. Now we describe a component of *Theorema* that is designed for domain-specific reasoning: It supports solving and computing in a special theory of differential equations. More precisely, it deals with linear two-point *boundary value problems* (BVPs). Before going into details, we give a practically relevant example that describes damped oscillations; see [50, p. 109] for details: Given a forcing function $f \in C^\infty[0, \pi]$, we want to find the uniquely determined function $u \in C^\infty[0, \pi]$ fulfilling $u'' + 2u' + u = f$ and $u(0) = u(1) = 0$.

The idea of our method is to reformulate this problem stated "on the functional level" as an equivalent problem posed "on the level of operators". On this level, it turns out that we can model the operators by noncommutative polynomials and solve for the relevant

operator (called the Green's operator) by a new symbolic technique. The result is then translated back to the functional level, where the solution is traditionally specified via the so-called Green's function g as $u = \int_0^1 g(x,\xi) f(\xi)\,\mathrm{d}\xi$. In the above example, one has

$$g(x,\xi) = \begin{cases} \frac{1}{\pi}(\pi - x)\xi\, \mathrm{e}^{\xi - x} & \text{if } 0 \leqslant \xi \leqslant x \leqslant \pi, \\ \frac{1}{\pi}(\pi - \xi)x\, \mathrm{e}^{\xi - x} & \text{if } 0 \leqslant x \leqslant \xi \leqslant \pi. \end{cases}$$

We will come back to this problem at the end of this subsection.

For the general problem formulation, let $[a,b]$ be a finite interval in $\mathbb{R}$ and T a linear differential operator with constant coefficients (the method has been extended to cover also operators with variable coefficients as described in [78], but we want to keep things simple in this presentation) given by $Tu = c_0 u^{(n)} + \cdots + c_{n-1}u' + c_n u$, where c_0 is nonzero. We view T as a linear operator on the vector space $C^\infty[a,b]$. The boundary operators $B_1, \ldots, B_n$ are defined on the same domain; for each $i = 1, \ldots, n$ we have $B_i u = p_{i,0}u^{(n-1)}(a) + \cdots + p_{i,n-1}u'(a) + p_{i,n}u(a) + q_{i,0}u^{(n-1)}(b) + \cdots + q_{i,n-1}u'(b) + q_{i,n}u(b)$, where the coefficients $p_{i,j}, q_{i,j}$ are real numbers. Now the BVP for T and $B_1, \ldots, B_n$ is to find for each forcing function $f \in C[a,b]$ a function $u \in C^n[a,b]$ such that

$$\begin{aligned} &Tu = f, \\ &B_1 u = \cdots = B_n u = 0. \end{aligned} \tag{i}$$

Since we have to find u in dependence on f, what we are really searching for is an operator G that maps each forcing function f to the corresponding solution u; such an operator is usually called the *Green's operator* of the BVP (i); see [84] for a detailed treatment. Note that we presuppose regular BVPs, meaning the solution u exists uniquely for each forcing function f. See Section 3.5 of [77] for some first results about nonregular BVPs.

The Green's operator can be defined analogously for many other types of BVPs for ODEs and PDEs, and it can often be described as an integral operator having a so-called *Green's function* g as its kernel. In the case of (i), this is indeed possible [25], leading to the Green's operator

$$Gf(x) = \int_a^b g(x,\xi) f(\xi)\,\mathrm{d}\xi. \tag{ii}$$

Thus one *can* reduce the search for the operator G to the search of the bivariate function g, and there is a solution method going along these lines [42]. However, working *directly on the operator level* seems more natural to us since the actual solution of any boundary value problem is always an operator no matter whether it is given through a kernel function (which is only possible for linear problems), so we have developed a new method for determining the Green's operator G in a suitable polynomial setting; see the journal article [77].

One crucial idea in our method is to model the key operators of differentiation, integration and boundary values as the indeterminates of a new polynomial ring whose multiplication should be interpreted as operator composition. Obviously this involves *noncommutative polynomials*. We need the following key operators as indeterminates: The differentiation $u \mapsto u'$ is represented by the indeterminate D, the antiderivative operator

$u \mapsto (x \mapsto \int_a^x u(\xi)\,\mathrm{d}\xi)$ by A, its dual $u \mapsto (x \mapsto \int_x^b u(\xi)\,\mathrm{d}\xi)$ by B, the left boundary operator $u \mapsto (x \mapsto u(a))$ by L, and the right counterpart $u \mapsto (x \mapsto u(b))$ by R. Moreover, we have a parametrized family of multiplication operators M_f representing $u \mapsto (x \mapsto f(x)u(x))$. The functions f are assumed to range over an algebra $\mathfrak{F}$ of functions; see [75].

Based on a given analytic algebra, we can now introduce the noncommutative polynomial ring $\mathfrak{An}(\mathfrak{F}) = \mathbb{C}\langle A, B, D, L, R, M_f \mid f \in \mathfrak{F}\rangle$, which we have called the ring of *analytic polynomials*.

The *algorithm for solving* a BVP of the type (i) proceeds in four phases:

(1) We compute a *projector* $P \in \mathfrak{An}(\mathfrak{F})$ onto the nullspace of T by using some trivial linear algebra on the fundamental system of T (the latter is typically presupposed when solving a BVP).
(2) Employing some Moore–Penrose theory [63], we reduce (i) to the right-inversion problem $GT = 1 - P$, which can be solved immediately by factoring the characteristic polynomial of T.
(3) We rewrite the resulting expression $(1 - P)T^{\blacklozenge}$ (with $T^{\blacklozenge} \in \mathfrak{An}$ being the right inverse) with respect to a carefully selected system of 36 polynomial equations (e.g., Fundamental Theorem of Calculus, product rule, integration by parts). More precisely, the noncommutative polynomials represented by the right-hand side of these equations form a noncommutative Gröbner basis; see [10,21].
(4) The result is a polynomial in $\mathfrak{An}(\mathfrak{F})$ in a normal form that allows to read off the Green's function (ii) immediately.

Note the transition of *special reasoners*: We start with a solving situation in the theory of inhomogeneous differential equations. Extracting the essential relations between the key operators, we move to a solving situation in the theory of noncommutative polynomials—a typical process of algebraization as described in [76]. Finally, the operator obtained through right inversion is normalized by a special rewrite system, this now being an instance of computing in the special theory of reducing modulo noncommutative polynomial ideals.

As an example, let us come back to the problem mentioned at the beginning of this section: solving the boundary value problem for the differential operator $T = D2 + 2D + 1$ for the boundary conditions $Lu = 0$ and $Ru = 0$ on the interval $[0, \pi]$. Using the *Theorema* command

$$\text{Compute}[\text{Green}[D2 + 2D + 1, \langle L, R\rangle, \text{by } \to \text{GreenEvaluator}]]$$

we get the output

$$\begin{aligned}(1 - \pi^{-1})&\lceil \mathrm{e}^{-x}x\rceil A\lceil \mathrm{e}^{x}\rceil - \lceil \mathrm{e}^{-x}\rceil A\lceil \mathrm{e}^{x}x\rceil + \pi^{-1}\lceil \mathrm{e}^{-x}x\rceil A\lceil \mathrm{e}^{x}x\rceil\\ &- \pi^{-1}\lceil \mathrm{e}^{-x}x\rceil B\lceil \mathrm{e}^{x}\rceil + \pi^{-1}\lceil \mathrm{e}^{-x}x\rceil B\lceil \mathrm{e}^{x}x\rceil.\end{aligned}$$

The multiplication operators M_f are denoted by $\lceil f\rceil$ for the sake of readability (in the input and output as well). Note that one can immediately read off the corresponding term $g(x, \xi)$ for the Green's function (ii), which is typically defined by a case distinction on $\xi < x$ and $\xi > x$: The summands with A go into the first case, those with B into the

second; the multiplication operators before A and B yield terms in x, those after yield terms in ξ. Proceeding in this way, one arrives immediately at the Green's function given in the beginning of the description of the method.

3.5. Automated prover for geometry

Geometric reasoning is another traditional mathematical activity that *Theorema* supports by providing a domain-specific reasoner. The *Theorema* geometry prover [73] is designed for constructive geometry problems. Besides the known proving methods such as Wu's characteristic set method [23,94], Gröbner Bases method [8,43,55], and area method [24] we have also included two new approaches: systematic exploration of geometric configurations and a new method for proving nontrivial geometry theorems involving order relations, which we will describe here.

As a first step in the proving process we visualize the geometry statement to be proved using the Mathematica graphical tools. The graphic representation can use either random or user-specified coordinates for the free points of the statement. A numerical check of the validity of the statement is performed for the actual coordinates of the points. To be able to use the proving methods, the problem has to be transformed from its external form into a specific internal form. When the algebraic methods are used we separate the coordinates into independent and dependent variables and find an appropriate coordinate system by a heuristic algorithm. The obtained polynomials are simplified as much as possible. When the area method is used the constructions have to be expressed using simpler constructions for which elimination lemmata exist.

The area method is very convenient for computing expressions involving geometric quantities relative to a specified construction. Using this method we can explore given geometric configurations [12,72]. Namely, starting from a knowledge base that specifies some constructions many theorems concerning parallel and perpendicular lines, segments with proportional length, and triangles with proportional areas are automatically obtained. Further constructions can be specified in a new knowledge base and the exploration may continue without recomputing the results already obtained. The results of the intermediate steps can be displayed on request. To prove geometry theorems that involve order relation (i.e., their algebraic forms contain polynomial inequalities besides polynomial equalities) we combined Collins's CAD algorithm with the area method. By this new method (Area-Cad method) we first compute the expressions involved in the inequalities using the area method. This way we obtain a new problem, equivalent to the original one, which is expressed only in terms of the independent points of the original constructions. Then, by applying the CAD method we obtain the result in a reasonable time even for rather complicated problems. Below we give an example on how the geometry prover proceeds.

Example 1. We want to prove the proposition: "If r is the radius of the incircle and R is the radius of the circumcircle of a triangle then $r \leqslant R/2$".

We prepare the following input to *Theorema*:

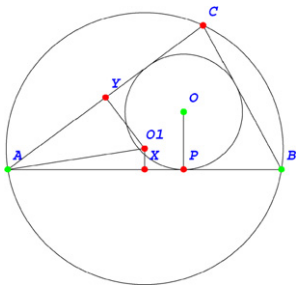


Fig. 4. Example problem for geometry prover.

Proposition["Tri", any[O, A, B, C, P, X, Y, O_1],

incircle[O, A, B, C, P] $\wedge$ midpoint[X, A, B] $\wedge$ midpoint[Y, A, C] $\wedge$
inter[O_1, tline[X, A, B], tline[Y, A, C]] $\wedge$ circle[O_1, A]].
$\Rightarrow 4 \cdot \text{seglength}[O, P]^2 \leqslant \text{seglength}[O_1, A]^2$

By the command

Simplify[Proposition["Tri"], by $\rightarrow$ GraphicSimplifier]

we obtain a graphical representation and a numerical check of the conclusion. For this configuration of the points (Fig. 4) *the relation* $4\overline{PO}^2 \leqslant \overline{AO_1}^2$ *holds. Now, we invoke the prover*:

Prove[Proposition["Tri"], by $\rightarrow$ GeometryProver,
ProverOptions $\rightarrow$ {Method $\rightarrow$ AreaCAD}].

The prover translates the proposition into a form that is expressed by special constructions. For these constructions the prover has built-in elimination lemmata. The lemmata are proved once and for all (*by elementary reasoning*) *and are used by the prover for the particular construction generated. We used* β_P, γ_P, α_P, *and* $\alpha 1_P$ *to denote the auxiliary points needed to express the construction for a point* P. *We get the following equivalent problem statement*:

$\{A, B, O\}$ free points
$\alpha_A B \perp AO, \quad \alpha_A \in AO$ (nondegenerate condition $A \neq O$)
$\alpha 1_A B \parallel B\alpha_A, \quad \dfrac{\overline{B\alpha 1_A}}{\overline{B\alpha_A}} = 2$ (ndg. cond. $B \neq \alpha_A$)
$\alpha_B A \perp BO, \quad \alpha_B \in BO$ (ndg. cond. $B \neq O$)
$\alpha 1_B A \parallel A\alpha_B, \quad \dfrac{\overline{A\alpha 1_B}}{\overline{A\alpha_B}} = 2$ (ndg. cond. $A \neq \alpha_B$)
$C = A\alpha 1_A \cap B\alpha 1_B$ (ndg. cond. $A \neq \alpha 1_A$, $B \neq \alpha 1_B$, $A\alpha 1_A \nparallel B\alpha 1_B$)
$PO \perp AB, \quad P \in AB$ (ndg. cond. $A \neq B$)

$$\alpha_C O \perp AC, \quad \alpha_C \in AC \qquad \text{(ndg. cond. } A \neq C\text{)}$$

$$XA \parallel AB, \quad \frac{\overline{AX}}{\overline{AB}} = \frac{1}{2} \qquad \text{(ndg. cond. } A \neq B\text{)}$$

$$YA \parallel AC, \quad \frac{\overline{AY}}{\overline{AC}} = \frac{1}{2} \qquad \text{(ndg. cond. } A \neq C\text{)}$$

$$Y\gamma_{O_1} \perp YA, \quad \frac{\overline{Y\gamma_{O_1}}}{\overline{YA}} = r_{23} \qquad \text{(ndg. cond. } Y \neq A\text{)}$$

$$X\beta_{O_1} \perp XA, \quad \frac{\overline{X\beta_{O_1}}}{\overline{XA}} = r_{22} \qquad \text{(ndg. cond. } X \neq A\text{)}$$

$$O_1 = X\beta_{O_1} \cap Y\gamma_{O_1} \qquad \text{(ndg. cond. } X \neq \alpha 1_A \beta_{O_1}, Y \neq \gamma_{O_1}, \beta_{O_1} X \nparallel \gamma_{O_1} Y\text{)}$$

with additional constraints $\overline{AB}^2 - \overline{AP}^2 > 0$, $\overline{AB}^2 - \overline{BP}^2 > 0$, $\overline{AC}^2 - \overline{A\alpha_C}^2 > 0$, *and* $\overline{AC}^2 - \overline{C\alpha_C}^2$ *imply that* $-\overline{AO_1}^2 + 4\overline{PO}^2 \leqslant 0$.

Next, the area method is used to obtain simpler forms of the constraints and of the conclusion. Finally, we get a new problem equivalent to the original one, expressed in terms of lengths of segments and oriented areas of triangles, which depends only on the free points. We rewrite the lengths of segments and areas of triangles using the coordinates of the points in a Cartesian coordinate system having the x-axes $\{A, O\}$. *Applying the CAD algorithm to this expression we obtain that the proposition is true.*

3.6. Verification of imperative programs

Verification of algorithms and programs is an important part of the theory exploration process. For instance, the user might want to verify algorithms or programs she developed before adding them into the library. They can be written in different styles. In this section we describe the *Theorema* tools to generate verification conditions for imperative programs. Similar tools for functional programs are described in Section 3.7.

In the *Theorema* system we provide a set of commands for defining imperative programs and reasoning about them [45]. The programming syntax is illustrated by the following example:

Program["Division", Div[$\downarrow x, \downarrow y, \uparrow rem, \uparrow quo$],

$quo := 0;\ rem := x;\ \ \text{while}[y \leqslant rem,\ rem := rem - y;\ quo := quo + 1]]$.

Further information on the program can be given in the "while" construct by the optional arguments "Invariant" and "TerminationTerm". Additionally, one may express the specification of the program in the usual *Theorema* syntax:

Specification["Division", Div[$\downarrow x, \downarrow y, \uparrow rem, \uparrow quo$],

$\text{Pre} \rightarrow ((x \geqslant 0) \wedge (y > 0))$,

$\text{Post} \rightarrow ((quo * y + rem = x) \wedge (0 \leqslant rem < y))$].

The verification condition generator that we provide uses for such programs Hoare Logic and the weakest precondition strategy [32]. The formulae produced are stored in a form directly understood by the reasoners of *Theorema*. Therefore, both the formulae and the proofs (generated by *Theorema*) are shown in a style meant to ease the understanding of correctness arguments. Failed proofs can give useful hints for modifying the program or the specification, or for adding appropriate knowledge. Furthermore, one can use *Theorema* reasoners with implicit knowledge about the used domain (see Section 3.1, for example). This makes proofs more compact and readable, in contrast to proving in pure predicate logic with explicit assumptions.

While the work outlined above is practical and experimental, we are making progress on a more challenging aspect by approaching the problem of *generating invariants* of while-loops. We believe that the effectiveness of automated verification of (imperative) programs is sensitive to the ease with which invariants, even trivial ones, can be (partially) automatically deduced, thus relieving the programmer (or the maintainer) of many tedious low-level tasks.

Our approach to solving this problem combines the weakest precondition strategy with combinatorial and algebraic methods for detecting properties of the variables modified in the loop. Although pioneered quite early in the community [33], this idea has not received much attention until recently in works investigating possible uses of Gröbner Bases techniques [74]. Our implementation [41,48] proceeds as follows: First, the recurrence equations expressing the values of the variables are extracted from the body of the loop. In the example, these are $quo_0 = 0$, $quo_{k+1} - quo_k = 1$, $rem_0 = x$, and $rem_{k+1} - rem_k = -y$, where k is a new variable representing the current iteration of the loop. Next, if the equations are independent (as in our example) or not mutually dependent, they are solved by geometric series manipulations or by the Gosper–Zeilberger algorithm; see, e.g., [36]. In the latter case we use the *Theorema* version of the Paule–Schorn implementation of this algorithm [64] to produce the closed-form $quo_k = 0 + k$, $rem_k = x - k * y$. We then eliminate k by a call to an appropriate routine, obtaining $rem = x - quo * y$ as an invariant for the loop. In addition to the generated invariants, there might be other invariant properties (linear inequalities, modular expressions, etc.) that still have to be given by the user. The generated and user-asserted invariants are then used together with other information obtained in the verification process to be able to apply the weakest precondition strategy. Moreover, from the explicit expressions of the variables, one is able to detect the termination term $rem - y$, and also to actually compute the number of iterations, by solving on k. If the equations are mutually dependent (as in a recursive program for computing the Fibonacci numbers), we apply a more sophisticated technique of generating functions [85] which is also able to generate the explicit expression of the values of the variables.

These techniques allow the automatic generation of loop invariants and termination terms for a large class of examples. We are currently investigating the extension of our system with techniques that use Gröbner Bases and with methods for handling nested loops [48].

3.7. Verification of recursive functional programs

We present here (on the basis of an example) a practical approach to the automatic generation of verification conditions for functional recursive programs, which complements the work on the synthesis of functional programs and on the verification of imperative programs described above.

Consider the following program schema:

$$F[x] = \begin{cases} S[x] & \Leftarrow\ Q[x] \\ C[x, F[R[x]]] & \Leftarrow \text{otherwise} \end{cases}$$

with the precondition $I_F[x]$ and the postcondition $O_F[x, y]$ as specification. To verify such a program we can use one of the theories that model the notion of computation; see [57] for a survey. However, in the context of automatic reasoning, one needs the theory of computation formalized as available knowledge for the used automatic reasoning system. The method presented here aims at generating first-order verification conditions which depend only on the knowledge relevant to the domain of the functions and predicates used in the program (we call this the *local theory*). The correctness proof of the method itself requires (only once) the use of a theory of computation—in our case the Scott fixpoint theory [29].

The first group of the generated verification conditions ensures that the inputs to each function satisfy the respective precondition:

$$\begin{aligned} &\underset{x:I_F[x]}{\forall}\big(Q[x] \Rightarrow I_S[x]\big) \qquad \underset{x:I_F[x]}{\forall}\big(\neg Q[x] \Rightarrow I_F[R[x]]\big) \\ &\underset{x:I_F[x]}{\forall}\big(\neg Q[x] \Rightarrow I_R[x]\big) \qquad \underset{x:I_F[x]}{\forall}\big(\neg Q[x] \Rightarrow \underset{y}{\forall}\big(O_F[R[x], y] \Rightarrow I_C[x, y]\big)\big). \end{aligned}$$

The second group of the generated verification conditions ensures that the produced output satisfies the postcondition of F:

$$\begin{aligned} &\underset{x:I_F[x]}{\forall}\big(Q[x] \Rightarrow O_F[x, S[x]]\big) \\ &\underset{x:I_F[x]}{\forall}\big(\neg Q[x] \Rightarrow \underset{y}{\forall}\big(O_F[R[x], y] \Rightarrow O_F[x, C[x, y]]\big)\big). \end{aligned}$$

In fact, using the Scott induction principle, one can prove that the conditions above are sufficient for the partial correctness of F, under the assumption that S, C, and R are totally correct; see [70]. Finally, the condition $\underset{x:I_F[x]}{\forall}(F' \downarrow x)$ ensures the totality of F, where F' is defined as

$$F'[x] = \begin{cases} 0 & \Leftarrow Q[x] \\ F'[R[x]] & \Leftarrow \text{otherwise} \end{cases}$$

and $F' \downarrow x$ means that F *terminates on* x and has to be expressed using the fixpoint theory of functions. Note that the totality condition is not expressed in the local theory alone. However, it only depends on Q and R, and, hence, can be used for an entire class of programs. We are studying the possibility of expressing this condition in the local theory (see [41]), as well as the application of this principle to more complex recursive schemata.

3.8. Interface to external systems

Theory exploration gives rise to different reasoning tasks that normally require the application of different techniques. Therefore it is handy to have access to several systems that are specialized in different, complementary reasoning methods. Besides its "internal" provers, solvers, and simplifiers, *Theorema* can use "external" automated reasoning systems via a special interface. The interface links *Theorema* with the external provers Bliksem [30], EQP [59], E [80], Gandalf [87], Otter [58], Scott [38], Setheo [56], Spass [90], Vampire [71], Waldmeister [6], and with the finite model and counterexample searcher Mace [60]. Scott, Setheo, and Waldmeister are linked to *Theorema* indirectly: The proving problem given in *Theorema* syntax is first translated into the TPTP format [86], and then, by the tptp2X converter, into the syntax of the external prover. The link with the other provers is direct, translating the proving problem from *Theorema* to the external system format without any intermediate routine. Indirect links are easy to establish while direct links are more flexible and give the user more control. The output of the external systems is, normally, not translated back to the *Theorema* syntax. (The only exception is the call to Otter.) Instead, the user is given the information whether the external system succeeded in finding a proof for the given problem, or, as in the case of calls to Mace, whether a countermodel was found. The output of the external provers can still be seen, in the respective prover's own format, by a click on a hyperlink in the *Theorema* proof notebook.[1]

The design of the interface allows combining various external systems with each other or with internal *Theorema* provers in a similar way the internal provers are combined with each other. From the user's point of view, within a *Theorema* session, there is no difference between calling an internal prover or an external system.

Besides the external deduction systems, *Theorema* is linked to TPTP [86] which is a comprehensive library of the automated theorem proving test problems that are available today. The TPTP2Theorema converter, written in Mathematica, translates the library problems into *Theorema* format. The converter works in an interactive mode. Upon calling, it opens a notebook with the description of steps necessary to convert TPTP problems into the *Theorema* format. The user has just to follow the corresponding links for each step. It is possible to translate the entire library, or separate files or directories. The translated problems are stored as Mathematica notebooks. The structure of such a notebook follows the structure of the original problem files, having sections for the header, theory (the axioms from the original file, the included axioms and assumptions) and conjectures. The notebooks also contain a title part with links to the TPTP web page and documentation; a description of the problem name, form and domain; a section with conjectures formulated in *Theorema* syntax, problem status explanation, and the corresponding Prove statement. In addition, a separate TPTP browser notebook is created. It shows the contents of translated TPTP library, with the directory and file names and hyperlinks to their locations, and brief explanations of each problem or axiom file. Detailed description of the interface can be found in [54].

[1] Notebooks are part of Mathematica front end. They are complete interactive documents combining text, tables, graphics, calculations, and other elements.

4. Organizational tools

This section gives an overview of new *Theorema* tools that help the user to organize the theory exploration process. These tools do not explicitly contribute to the reasoning power of the system, but drastically improve its usability. We describe here the focus windows technique for proof presentation, the label management tool to organize knowledge bases, and "logicographic symbols" tool that allows the user to introduce arbitrary new mathematical symbols.

4.1. Focus windows

Understanding the outcome of a reasoner is an important step in theory exploration. For this reason (from the outset), *Theorema* emphasized attractive proof presentation. *Theorema* proofs are designed to resemble proofs done by humans, i.e., they contain formulae and explanatory text in English. Usually in textbooks, mathematical proofs are presented as linear sequences of proof steps. In long proofs the formulae used in a proof step occur, typically, a couple of lines, paragraphs, or even pages distant from the place in the text where the proof step is executed. Reference to the formulae used is traditionally done by labels and the reader has to jump back and forth between the formulae referenced and the proof step in which they are needed. This is unpleasant and makes understanding of proofs quite difficult even when the proofs are nicely structured and well presented.

Theorema provides various tools to help the reader browse the proofs: nested brackets at the right-hand window margin make it possible to contract entire subproofs to just one line; various color codes distinguish (temporary) proof goals from formulae in the (temporary) knowledge base; references to formulae are hyperlinks which will display the formulae referenced in small auxiliary windows; etc. By using hyperlinks as references to formulae, the readers of *Theorema* proofs can avoid back and forth jumps in the proof to understand the validity of a specific step. Still, reading and understanding linear proofs may be difficult even with these tools and similar tools (e.g., LΩui [82]).

Focus windows provide a new proof presentation technique to overcome this problem. This technique can be viewed as a systematic extension of the idea of using hyperlinks to reference formulae. It can be implemented for any proof assistant system that uses formal objects for proofs, i.e., a data structure that contains information on which formulae are used and which are produced in a given step, for each step in the proof. This means that also systems that do proof checking could make use of this technique. We emphasize that *Theorema* 's focus windows technique is *not* a reasoning tool. It is a presentation method: the successful or unsuccessful proof attempts are showed to the user in a style that helps her gain mathematical insight and knowledge.

The idea of the focus windows technique is simple but quite efficient: Given a proof step in a proof, the focus windows tool analyzes which formulae are used and which are produced in the respective step (the "relevant" formulae). Correspondingly, a window is composed that shows exactly these formulae. The window also contains buttons for moving to and analyzing the next or previous step in the proof. For the steps that branch to two or more subproofs the subsequent windows are displayed in contracted form and the user can decide which one to open next. The focus windows method allows to study proofs in

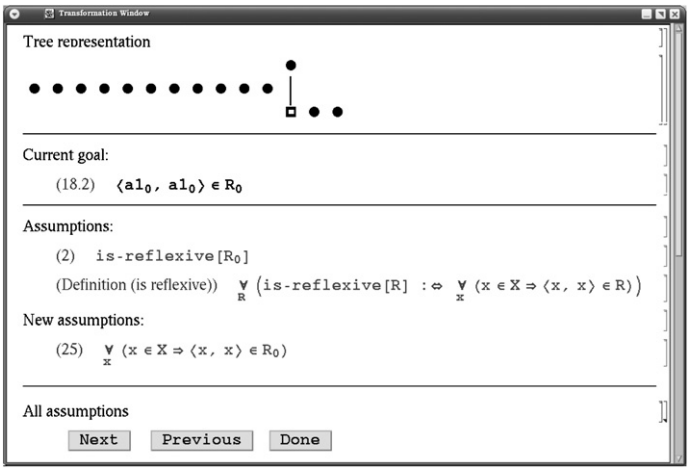


Fig. 5. A focus window.

a stepwise manner. Each step of the proof is shown to the user in two phases: an attention phase and a transformation phase. The focus window corresponding to an attention phase (the Attention Window) does not show the user the formulae inferred at the inspected proof step. These are shown in the focus window after the transformation phase (the Transformation Window, see example in Fig. 5). This window has

- a "goal area" in which the current goals are shown,
- an "assumptions area" in which the "relevant" assumptions are shown,
- a "proof tree area" in which the entire proof tree is displayed in a schematic, simplified form,
- an area that presents all the assumptions that are available (the "all assumptions area"),
- and a "navigation area" that helps the user navigate in the proof by clicking on various buttons.

The focus windows presentation technique can be used also for incomplete or incorrect proofs, making it also a useful tool for prover debugging. Details of the technique can be found in [67].

4.2. Label management

Theory exploration usually involves a large number of formulae. In the build-up of completely formalized mathematical knowledge bases, the systematic design and processing of structured labels (i.e., individual labels like "(1)", "(2)" or "(associativity)", etc., hierarchical section headings, key words like "definition" and "theorem", names of files, etc.) becomes vital to the automated structuring and restructuring of collections of formulae as input to formal reasoning tools like provers, simplifiers, algorithm verifiers, model checkers, etc. Consequently, we need algorithmic tools that handle all types of labels and

allow us to partition and combine, structure and restructure mathematical knowledge bases according to the structural information provided by the hierarchical labels.

We emphasize that, in our view, labels do *not* intend to have any logical meaning or functionality. This is in contrast to the goal of "annotations", etc. as, for example, in [22, 46,79], which convey at least part of the semantics. In our view, the semantics of formulae (in particular predicate logic formulae) is exclusively defined by their inclusion into the context of collection of other formulae (mathematical knowledge bases). The functionality of labels is purely organizational.

The *Theorema* system provides tools for the automatic assignment of labels to formulae and collections of formulae which are stored into notebooks created with respect to a set of few, simple, and intuitive rules. We call such notebooks "*Theorema* notebooks". Furthermore, the users can identify and combine, in various ways, mathematical knowledge stored in *Theorema* notebooks, without going into the "semantics" of the formulae [67–69]. The labels assigned to knowledge accumulated in a *Theorema* notebook are automatically generated in a hierarchical way. From the information provided by the user (section headings, notebook title, etc.) the tool automatically generates composite labels for each section, subsection, etc., and individual formula in the notebook. These composite labels are generated in three variants which we call long, short, and decimal composite labels, respectively.

For example, Fig. 6 shows a *Theorema* notebook that collects formulae expressing knowledge about length of tuples. The formulae in this notebook are part of the theory

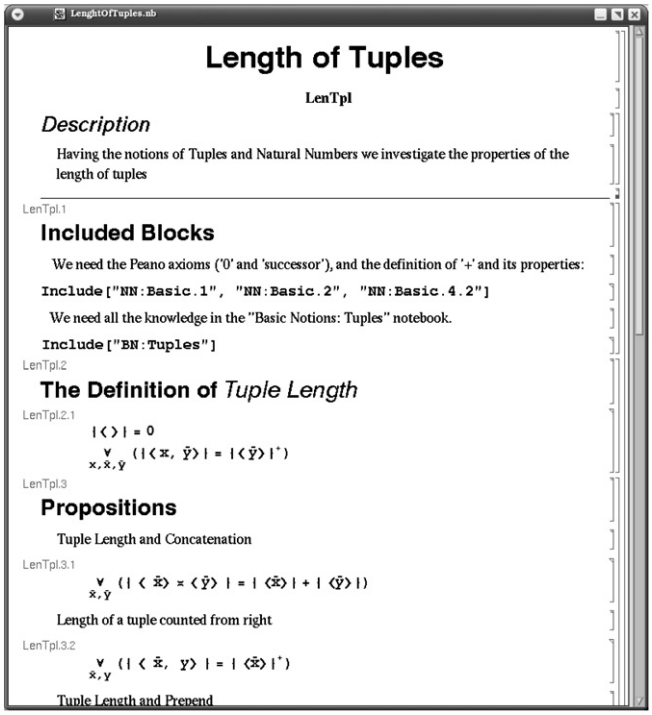


Fig. 6. A *Theorema* notebook.

exploration about tuples. Individual formulae and groups of formulae have certain labels attached to them. In the figure we can see their digital variant: the group of formulae with the header "Propositions" obtain automatically the label "LenTpl.3", while the individual formulae in the group obtain the labels "LenTpl.3.1", "LenTpl.3.2", etc. The long variant of, e.g., "LenTpl.2.1" is "LenTpl.The Definition of TupleLength.1". The user can also assign explicit labels to (groups of) formulae. The notebook in Fig. 6 also refers to (parts of) other *Theorema* notebooks, namely parts of the *Theorema* notebook collecting formulae about natural numbers (Include["NN:Basic.1", "NN:Basic.2", "NN:Basic.4.2"]) and the notebook that contains basic notions about tuples (Include["BN:Tuples"]).

The label management tools operate on libraries (collections) of *Theorema* notebooks. The tools are realized such that labels assigned to (groups of) formulae are guaranteed to be unique within a library of *Theorema* notebooks. Knowledge stored in *Theorema* notebooks can now be referenced by labels and used, for example, for calling reasoners:

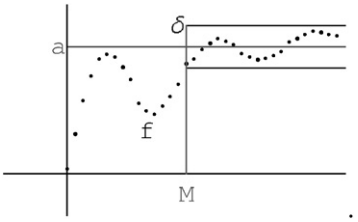
$$\text{Prove}[\text{“LenTpl.3.1”},\ \text{using} \to \langle \text{“BN:Tuples”}, \text{“NN:Basic”} \rangle,\ \text{by} \to \cdots].$$

4.3. Logicographic symbols

Two-dimensional syntax in *Theorema* is very flexible: By the use of the front end of the Mathematica system, which is the programming environment for the implementation of *Theorema*, programmable syntax comes for free (to a certain extent). However, the arsenal of mathematical symbols is limited by what Mathematica offers. The new *Theorema* tool of "logicographic symbols" goes a significant step further: With this tool we are now able to design any new symbol—even complicated ones—with arbitrary arity and slots for arguments at arbitrary position in a two-dimensional grid. These symbols can be nested to arbitrary depth. With an appropriate design, these symbols may convey the intuitive meaning of mathematical concepts. As an example, take the notion of limit$[f, a, \delta, M]$ (or similar notions that occurred in previous sections). This notion can be defined in *Theorema* by

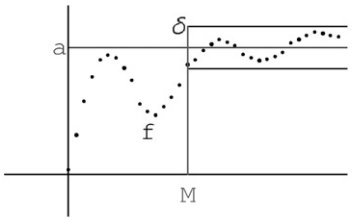
$$\text{limit}[f, a, \delta, M] \Leftrightarrow \underset{n}{\forall}\big(n > M \Rightarrow \big|f[n] - a\big| < \delta\big).$$

In *Theorema*, we may now design a new graphical symbol for this notion by using the graphics tools of Mathematica (or just by taking a hand-drawing and making it a Mathematica object) and equip it with slots (boxes) for the arguments, like, for example, the new symbol



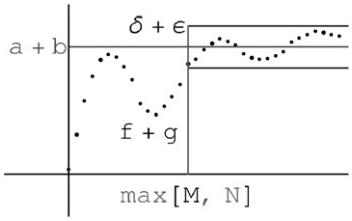
Now, by the following declaration

Logicographic-Declaration["limit symbol", any[f, a, δ, M],



$\leftrightharpoons$ limit[f, a, δ, M]]

this graphical symbol can interchangeably be used instead of the 4-ary predicate symbol "limit" and will now be available for both input and output of formulae (for example within proofs generated by the *Theorema* provers). Internally, formulae with such "logicographic symbols" are just ordinary *Theorema* (i.e., predicate logic) formulae with the logicographic symbols replaced by the respective function or predicate constants that appear in the declaration. For example, one can substitute arbitrary terms for the four arguments of the symbol:



In other words, formulae with logicographic symbols are completely "logical" as far as their meaning within a mathematical knowledge base is concerned but, at the same time, they also convey the intuitive ("graphical") meaning of the formulae. Hence the name "logicographic". Although logicographic symbols do not add anything to the logical expressiveness of the system, they may enhance readability and ease understanding of mathematical texts significantly. The implementation in the frame of *Theorema*, together with examples and other tools for enhancing the readability of formulae (e.g., a method of shading two-dimensional subformulae of formulae instead of using nested parentheses) is described in [62].

5. Related work

In this section we first compare Lazy Thinking with other approaches to program/algorithm synthesis. Next, we give a brief overview of some other systems that, like *Theorema*, are designed as mathematical assistants.

The survey paper [3] considers three methods of program synthesis: constructive/deductive, schema-based, and inductive synthesis. Lazy thinking is similar to the deductive synthesis, which uses deduction to synthesize programs by solving unknowns during the applications of rules. In the context of inductive proof planning in [49], a proof of the correctness of an algorithm is set up, and the unknown parts of the algorithm are replaced

with metavariables, which will be instantiated as the proof planning progresses. Proof critics [39] are used to overcome failure of proofs and generate new lemmata, by analyzing the failure of rippling (a method to guide rewriting that tries to eliminate the structural differences between the induction hypothesis and the induction conclusion). Lazy thinking is similar to this approach in that it also attempts to prove the correctness conjecture, and it uses the failure of the proof to generate conjectures and complete the proof. However, Lazy Thinking takes into consideration failing proof situations (temporary assumptions and current goal) to generate its conjectures. In the context of Lazy Thinking the algorithm schemata are similar to those in the schema-based synthesis methods, with a template that captures the flow of the program and specifications (constraints) on the ingredients of the template.

In [34] the authors use a notion of generic correctness of schemata (modulo correctness of subalgorithms in the schema)—steadfastness—and programs are synthesized by transforming steadfast schemata into correct programs. Similarly, the preprocessing of Lazy Thinking, as described in [18], ensures a notion of correctness of a schema.

One of the most successful approaches to schema-based synthesis is that of Smith [83], who uses a category theory framework to represent schemata and transformations. This setting ensures that transformations are correct. Moreover, a large library (hierarchy) of algorithm schemata is available and used to guide the synthesis. Preprocessing Lazy Thinking gives a similar transformational flavor to our method. It allows to take offline some of the more difficult proof obligations: We apply once and for all Lazy Thinking and then we just have to show that the concrete problem and algorithm schema selected are instances of a problem and an algorithm schema that have been preprocessed.

A distinctive feature of the Lazy Thinking method (for algorithm synthesis) is that it is applied in the context of systematic, computer-supported theory exploration, being one of the tools available for theory exploration. Therefore, the programs/algorithms are expressed in the *Theorema* language frame and the implementation of the Lazy Thinking mechanism is integrated with the Prove-call of *Theorema*.

As a mathematical assistant, *Theorema* shares its research goals with systems designed for supporting formalization of mathematics, like Coq [4], HELM [1], Mizar [88], Nuprl [27], and Ωmega [81], just to name a few. Of them the Ωmega system is closest to *Theorema*. Ωmega is a mixed-initiative system with the ultimate purpose of supporting theorem proving in mathematics and mathematics education. It contains a proof-planner based on an extended STRIPS algorithm. Furthermore, like *Theorema*, Ωmega aims at integrating computer algebra support into proving. In the integration Ωmega uses the mathematical knowledge implicit in the computer algebra system to extract proof plans that correspond to the mathematical computation in the computer algebra system. The proof-search engine in Ωmega uses *methods* which fully specify the input/output behavior of the tactics they are associated with, to the point where it is possible to automatically generate the tactic from its specification (method). This is a very elegant approach, which has the advantage that the proof-planning mechanism may manipulate methods (i.e., adapting general ones to special mathematical domains) and the corresponding tactics will then be automatically generated from the methods thus "synthesized". However, the situation in *Theorema* is quite different: grouped inference rules have a heuristic of their own, which can be modified in order to enable them to handle specific classes of proofs. They perform the parts

of the proof which "they know how to handle". Hence, *Theorema* emphasizes more on automated proving while Ωmega basically proceeds with goal transformation using tactics. They differ also on the type systems of the underlying logic that is untyped in *Theorema*, and has decidable nondependent types in Ωmega.

Coq and Nuprl implement variants of intuitionistic type theory: calculus of inductive constructions in Coq, and computational type theory in Nuprl. Both systems have a proof checking kernel, and use tactics to transform goals. Coq and Nuprl, based on constructive logic, are very well suited for reasoning about computation because they provide as primitive notions ways of constructing primitive recursive functions. However, doing mathematics in such a system is most of the time quite different from the mathematics one reads in textbooks. Both Coq and Nuprl come with a large mathematical library. *Theorema* in its own does not have such a large collection, but it can access the algorithm library of Mathematica and the proving problem library TPTP.

As a proof-checking system, Mizar offers mathematicians the possibility to develop theories by defining new concepts and proving theorems in a strictly formalized manner, where each step of the formalization is checked by the system. It has the largest library by far: more than 40 thousand theorems.

The HELM project aims at creating electronic library of mathematics. It tries to integrate the current tools for the automation of formal reasoning and the mechanization of mathematics (proof assistants and logical frameworks) with the most recent technologies for the development of web applications and electronic publishing, eventually passing through XML. The final goal is the development of a suitable technology for the creation and maintenance of a virtual, distributed, hypertextual library of formal mathematical knowledge.

A more detailed comparison of systems for formalization of mathematics can be found in [91] where fifteen such systems are compared: HOL, Mizar, PVS, Coq, Otter, Isabelle, Agda, ACL2, PhoX, IMPS, Metamath, *Theorema*, Lego, Nuprl, and Ωmega.

6. Conclusion and future work

We presented our view on mathematical theory exploration and described methods and tools developed in the *Theorema* project to assist mathematicians in exploring theories. As an example of a method we presented Lazy Thinking that is used in the algorithm synthesis stage of theory exploration. The tools comprise reasoners (provers, simplifiers, solvers) for general and special theories, and the tools to organize and reorganize the exploration process. We gave a general overview, details can be found in the cited publications of the *Theorema* group.

Currently we are working on a major redesign of the *Theorema* system that takes into account the experience we gained by using the system for the experimental exploration of various theories. A main lesson we learned from this is that typical users of *Theorema* ("working mathematicians") do not only want to use the reasoners of the system as "black boxes" (although most of these reasoners provide various options to influence the way they work on concrete problems). Rather, typical users want to modify or extend the available reasoners or even want to implement their own ideas for computer-supported reasoning

while they are working on the exploration of particular theories. For making this possible, the code of the reasoners must be open and, ideally, they should be programmed in the same language in which the mathematical theories are presented. Moreover, it also should be possible to prove the correctness of new reasoners within the system. This means that the object language should be represented in the meta-language, i.e., in the case of *Theorema*, in (the *Theorema* version of) predicate logic. It is clear that this needs the implementation of a kind of logical "reflection". Theoretically, it is known how this can be done; see, e.g., [37]. However, it is a nontrivial task to provide reflection in an attractive and user-friendly way that allows to migrate easily between object and meta level during a theory exploration session. We did not yet find a satisfactory answer for this problem.

At the same time, we are undertaking a couple of major case studies of theory exploration: the build-up of a verified knowledge base for Gröbner Bases theory that combines the nonalgorithmic and algorithmic aspects of the theory; a systematic exploration of the theory of Hilbert spaces of which the work on the symbolic solution of differential equations described in this paper is only one part; the exploration of the theory of tuples for which the work on the algorithmic synthesis of sorting algorithms is a first step; and using *Theorema*, in particular the PCS and S-decomposition provers, in the undergraduate calculus education in analysis. In order to avoid misunderstandings we want to emphasize that the goal of the *Theorema* system is computer support for the (95% of) "easy" reasoning during the exploration of mathematical theories and *not* the automated invention of "difficult" or ingenious points in a theory or in a proof. Of course, the notion of "easy" and "difficult" is relative: What seemed difficult twenty years ago, by new reasoning techniques, is easy now and what seems to be difficult now may become easy in twenty years' time. For example, the new method in Section 3.4, based on noncommutative Gröbner Bases, allows to "invent" Green theorems just by one computation modulo a Gröbner basis for operator equalities, where each of the inventions needed human ingenuity so far.

Acknowledgements

Theorema was supported by the Austrian Science Foundation (FWF) under the projects SFB F1302 and SFB F1322, the EU "Calculemus" project (HPRN-CT-2000-00102), the regional government of Upper Austria under the "Prove" project, and the Johannes Kepler University of Linz under the "CreaComp" project. The program verification project within *Theorema* is supported by the Austrian Ministry of Education, Science, and Culture, the Austrian Ministry of Economy and Work, and by the Romanian Ministry of Education and Research in the frame of the project "e-Austria Timişoara".

References

[1] A. Asperti, L. Padovani, C. Sacerdoti Coen, F. Guidi, I. Schena, Mathematical knowledge management in HELM, Ann. Math. Artificial Intelligence 38 (1) (2003) 27–46.

[2] L. Bachmair, N. Dershowitz, D. Plaisted, Completion without failure, in: H. Aït-Kaci, M. Nivat (Eds.), Resolution of Equations in Algebraic Structures, vol. 2, Elsevier Science, Amsterdam, 1989, pp. 1–30.

[3] D. Basin, Y. Deville, P. Flener, A. Hamfelt, J.F. Nilsson, Synthesis of programs in computational logic, in: M. Bruynooghe, K.-K. Lau (Eds.), Program Development in Computational Logic, in: Lecture Notes in Comput. Sci., vol. 3049, Springer, Berlin, 2004, pp. 30–65.
[4] Y. Bertot, P. Castèran, Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions, Texts in Theoretical Computer Science, An EATCS Series, Springer, Berlin, 2004.
[5] W. Bledsoe, Challenge problems in elementary analysis, J. Automat. Reason. 6 (1990) 341–359.
[6] A. Buch, T. Hillenbrand, Waldmeister: Development of a high preformance completion based theorem prover, SEKI-Report SR-96-01, University of Kaiserslautern, Germany, 1996.
[7] B. Buchberger, An algorithm for finding the basis elements in the residue class ring modulo a zero dimensional polynomial ideal, PhD thesis, Mathematical Institute, University of Innsbruck, Austria, 1965, in German.
[8] B. Buchberger, Gröbner-bases: An algorithmic method in polynomial ideal theory, in: N.K. Bose (Ed.), Multidimensional Systems Theory, Reidel Publishing Company, Dordrecht, 1985, pp. 184–232.
[9] B. Buchberger, Symbolic computation: Computer algebra and logic, in: F. Baader, K.U. Schulz (Eds.), Frontiers of Combining Systems, in: Applied Logic Series, vol. 3, Kluwer Academic Publishers, Dordrecht, 1996, pp. 193–219.
[10] B. Buchberger, Introduction to Gröbner bases, in: [21], pp. 3–31.
[11] B. Buchberger, Theory exploration versus theorem proving, in: A. Armando, T. Jebelean (Eds.), Proc. of Calculemus'99, in: ENTCS, vol. 23, Elsevier, Amsterdam, 1999.
[12] B. Buchberger, Theory exploration with Theorema, Analele Universităţii din Timişoara, Ser. Matematică-Informatică 38 (2) (2000) 9–32.
[13] B. Buchberger, Algorithm invention and verification by Lazy Thinking, in: [65], pp. 2–26.
[14] B. Buchberger, Algorithm-supported mathematical theory exploration: A personal view and strategy, in: [16], pp. 236–250.
[15] B. Buchberger, Towards the automated synthesis of a Gröbner bases algorithm, RACSAM (Review of the Spanish Royal Academy of Sciences) 98 (1) (2004) 65–75.
[16] B. Buchberger, J.A. Campbell (Eds.), Proc. of the 7th Int. Conf. on Artificial Intelligence and Symbolic Computation, AISC'04, Hagenberg, Austria, Lecture Notes in Artif. Intell., vol. 3249, Springer, Berlin, 2004.
[17] B. Buchberger, A. Crăciun, Algorithm synthesis by Lazy Thinking: Examples and implementation in Theorema, in: F. Kamareddine (Ed.), Proc. of the Mathematical Knowledge Management Workshop, Edinburgh, in: ENTCS, vol. 93, Elsevier, Amsterdam, 2003, pp. 24–59.
[18] B. Buchberger, A. Crăciun, Algorithm synthesis by Lazy Thinking: Using problem schemes, in: [66], pp. 90–106.
[19] B. Buchberger, C. Dupré, T. Jebelean, F. Kriftner, K. Nakagawa, D. Văsaru, W. Windsteiger, The Theorema project: A progress report, in: [44], pp. 98–113.
[20] B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuţa, D. Văsaru, A survey of the Theorema project, in: W. Küchlin (Ed.), Proc. of the Int. Symposium on Symbolic and Algebraic Computation ISSAC'97, ACM Press, New York, 1997, pp. 384–391.
[21] B. Buchberger, F. Winkler, Gröbner Bases and Applications, Cambridge University Press, Cambridge, UK, 1998, Proc. of the Int. Conf. "33 Years of Gröbner Bases", RISC, Austria, London Mathematical Society Lecture Note Series, vol. 251, 1998.
[22] O. Caprotti, D. Carlisle, OpenMath and MathML: Semantic mark up for mathematics, ACM Crossroads 6 (2) (1999), special issue on Markup Languages.
[23] S.C. Chou, Mechanical Geometry Theorem Proving, Reidel, Dordrecht, Boston, 1975.
[24] S.C. Chou, X.S. Gao, J.Z. Zhang, Automated production of traditional proofs in euclidian geometry, in: Proc. of 8th IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, 1993, pp. 48–56.
[25] E.A. Coddington, N. Levinson, Theory of Ordinary Differential Equations, McGraw-Hill Book Company, New York, 1955.
[26] G.E. Collins, Quantifier elimination for real closed fields by cylindrical algebraic decomposition, in: Second GI Conf. on Authomata Theory and Formal Languages, in: Lecture Notes in Comput. Sci., vol. 33, Springer, Berlin, 1975, pp. 134–183.
[27] R. Constable, Implementing Mathematics Using the Nuprl Proof Development System, Prentice-Hall, Englewood Cliffs, NJ, 1986.

[28] A. Crăciun, B. Buchberger, Preprocessed Lazy Thinking: Synthesis of sorting algorithms, Technical Report 04-17, RISC, Linz, Austria, 2004.
[29] J. W. de Bakker, D. Scott, A theory of programs, in: IBM Seminar, Vienna, Austria, 1969.
[30] H. de Nivelle, Bliksem resolution prover. Available to download from http://www.mpi-sb.mpg.de/~nivelle/software/bliksem/, 1999.
[31] J. Denzinger, S. Schulz, Analysis and representation of equational proofs generated by a distributed completion based proof system, SEKI-Report SR-94-05, University of Kaiserslautern, Germany, 1994.
[32] E.W. Dijkstra, A Discipline of Programming, Prentice-Hall, Englewood Cliffs, NJ, 1976.
[33] B. Elspas, M.W. Green, K.N. Lewitt, R.J. Waldinger, Research in interactive program—proving techniques, Technical Report, Stanford Research Institute, Menlo Park, CA, USA, May 1972.
[34] P. Flener, K.-K. Lau, M. Ornaghi, J.D.C. Richardson, An abstract formalization of correct schemas for program synthesis, J. Symbolic Comput. 30 (1) (2000) 93–127.
[35] H. Ganzinger (Ed.), Proc. of the 16th Int. Conf. on Automated Deduction, CADE'99, Trento, Italy, Lecture Notes in Artif. Intell., vol. 1632, Springer, Berlin, 1999.
[36] R.W. Gosper, Decision procedures for indefinite hypergeometric summation, Proc. Nat. Acad. Sci. USA 75 (5–6) (1978) 40–42.
[37] J. Harrison, Metatheory and reflection in theorem proving: A survey and critique, Technical Report CRC-053, SRI Cambridge, UK, 1995.
[38] K. Hodgson, J. Slaney, Semantic guidance for saturation-based theorem proving, TR-ARP 04-2000, Automated Reasoning Project, Australian National University, Canberra, Australia, 2000.
[39] A. Ireland, A. Bundy, Productive use of failure in inductive proof, J. Automat. Reason. 16 (1–2) (1996) 79–111.
[40] T. Jebelean, Natural proofs in elementary analysis by S-decomposition, Technical Report 01-33, RISC, Linz, Austria, 2001.
[41] T. Jebelean, L. Kovács, N. Popov, Experimental program verification in Theorema, in: Proc. of the 1st Int. Symposium on Leveraging Applications of Formal Methods, ISOLA'04, 2004.
[42] E. Kamke, Differentialgleichungen und Lösungsmethoden, vol. 1, tenth ed., Teubner, Stuttgart, 1983.
[43] D. Kapur, Using Gröbner bases to reason about geometry problems, J. Symbolic Computation 2 (1986) 399–408.
[44] M. Kerber, M. Kohlhase (Eds.), Proc. of Calculemus'2000, St. Andrews, UK, 2000.
[45] M. Kirchner, Program verification with the mathematical software system Theorem, Technical Report 99-16, RISC, Linz, Austria, 1999.
[46] M. Kohlhase, OMDoc: An infrastructure for OpenMath content dictionary information, ACM SIGSAM Bull. 34 (2) (2000) 43–48.
[47] B. Konev, T. Jebelean, Using meta-variables for natural deduction in Theorema, in: [44], pp. 160–175.
[48] L. Kovács, T. Jebelean, Automated generation of loop invariants by recurrence solving in Theorema, in: [66].
[49] I. Kraan, D. Basin, A. Bundy, Middle-out reasoning for synthesis and induction, J. Automat. Reason. 16 (1–2) (1996) 113–145.
[50] A.M. Krall, Applied Analysis, D. Reidel Publishing Company, Dordrecht, 1986.
[51] T. Kutsia, Equational prover of Theorema, in: R. Nieuwenhuis (Ed.), Proc. of the 14th Int. Conf. on Rewriting Techniques and Applications, RTA'03, Valencia, Spain, in: Lecture Notes in Comput. Sci., vol. 2706, Springer, Berlin, 2003, pp. 367–379.
[52] T. Kutsia, Solving equations involving sequence variables and sequence functions, in: [16], pp. 157–170.
[53] T. Kutsia, B. Buchberger, Predicate logic with sequence variables and sequence function symbols, in: A. Asperti, G. Bancerek, A. Trybulec (Eds.), Proc. of the 3rd Int. Conf. on Mathematical Knowledge Management, in: Lecture Notes in Comput. Sci., vol. 3119, Springer, Berlin, 2004, pp. 205–219.
[54] T. Kutsia, K. Nakagawa, An interface between Theorema and external automated deduction systems, Technical Report 00-29, RISC, Linz, 2000.
[55] B. Kutzler, S. Stifter, On the application of Buchberger's Algorithm to automated geometry theorem proving, J. Symbolic Comput. 2 (1986) 389–397.
[56] R. Letz, J. Schumann, S. Bayerl, W. Bibel, Setheo: A high-performance theorem prover, J. Automat. Reason. 8 (2) (1992) 183–212.
[57] J. Loeckx, K. Sieber, The Foundations of Program Verification, second ed., Teubner, Stuttgart, 1987.

[58] W. McCune, Otter 3.0 reference manual and guide, ANL-TR 94/6, Argonne National Laboratory, Argonne, USA, 1994.
[59] W. McCune, EQP. Available from http://www.mcs.anl.gov/AR/eqp/, 1999.
[60] W. McCune, Mace 2.0 user manual and guide, Technical memorandum ANL/MCS-TM 249, Argonne National Laboratory, Argonne, USA, 2001.
[61] E. Mellis, J. Siekmann, Knowledge-based proof planning, Artificial Intelligence 115 (1) (1999) 65–105.
[62] K. Nakagawa, Supporting user-friendliness in the mathematical software system Theorema, PhD thesis, RISC, Johannes Kepler University, Linz, Austria, 2002.
[63] M.Z. Nashed (Ed.), Generalized Inverses and Applications, Proc. of an Advanced Seminar Sponsored by the Mathematics Research Center, Academic Press, New York, 1976.
[64] P. Paule, M. Schorn, A Mathematica version of Zeilberger's algorithm for proving binomial coefficient identities, J. Symbolic Comput. 20 (5–6) (1995) 673–698.
[65] D. Petcu, V. Negru, D. Zaharie, T. Jebelean (Eds.), Proc. of SYNASC'03, 5th Int. Workshop on Symbolic and Numeric Algorithms for Scientific Computing, Timişoara, Romania, Mirton, 2003.
[66] D. Petcu, V. Negru, D. Zaharie, T. Jebelean (Eds.), Proc. of SYNASC'04, 6th Int. Symposium on Symbolic and Numeric Algorithms for Scientific Computing, Timişoara, Romania, Mirton, 2004.
[67] F. Piroi, Tools for using automated provers in mathematical theory exploration, PhD thesis, RISC, Johannes Kepler University, Linz, Austria, August 2004.
[68] F. Piroi, B. Buchberger, An environment for building mathematical knowledge libraries, in: C. Benzmüller, W. Windsteiger (Eds.), Proc. of the first Workshop on Computer-Supported Mathematical Theory Development, IJCAR'04, Cork, Ireland, 2004, pp. 19–29.
[69] F. Piroi, B. Buchberger, Label management in mathematical theories, Technical Report 2004-16, RICAM, Linz, Austria, 2004.
[70] N. Popov, Verification of simple recursive programs: Sufficient conditions, Technical Report 04-06, RISC, Linz, Austria, 2004.
[71] A. Riazanov, A. Voronkov, Vampire, in: [35], pp. 292–296.
[72] J. Robu, Systematic exploration of geometric configurations using Theorema based on Mathematica, in: Proc. of 3rd Int. Workshop on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC'01, Timişoara, Romania, 2001, pp. 209–216.
[73] J. Robu, Automated geometric theorem proving, PhD thesis, RISC, Johannes Kepler University, Linz, Austria, 2002.
[74] E. Rodriguez-Carbonell, D. Kapur, Automatic generation of polynomial loop invariants: Algebraic foundations, in: Proc. of the Int. Symposium on Symbolic and Algebraic Computation, ISSAC'04, Santander, Spain, ACM Press, New York, 2004, pp. 266–273.
[75] M. Rosenkranz, A polynomial approach to linear boundary value problems, PhD thesis, RISC, Johannes Kepler University, Linz, Austria, September 2003.
[76] M. Rosenkranz, The algorithmization of physics: Math between science and engineering, in: [16], pp. 1–8, invited talk.
[77] M. Rosenkranz, A new symbolic method for solving linear two-point boundary value problems on the level of operators, J. Symbolic Comput. 39 (2) (2005) 171–199.
[78] M. Rosenkranz, B. Buchberger, H.W. Engl, A symbolic algorithm for solving two-point BVPs on the operator, SFB Report 2003-41, Johannes Kepler University, Linz, Austria, 2003.
[79] P. Sandhu, The MathML Handbook, Charles River Media, 2002.
[80] S. Schulz, E—a brainiac theorem prover, J. AI Comm. 15 (2/3) (2002) 111–126.
[81] J. Siekmann, C. Benzmüller, Omega: Computer supported mathematics, in: S. Biundo, T. Frühwirth, G. Palm (Eds.), Proc. of the 27th German Conf. on Artificial Intelligence, KI'04, Ulm, Germany, in: Lecture Notes in Comput. Sci., vol. 3238, Springer, Berlin, 2004, pp. 3–28.
[82] J. Siekmann, S. Hess, C. Benzmüller, L. Cheikhrouhou, A. Fiedler, H. Horacek, M. Kohlhase, K. Konrad, A. Meier, E. Melis, M. Pollet, V. Sorge, LOUI: Lovely Omega User Interface, Formal Aspects of Computing 11 (3) (1999) 326–342.
[83] D.R. Smith, Mechanizing the development of software, in: M. Broy, R. Steinbrueggen (Eds.), Calculational System Design, Proc. of the NATO Advanced Study Institute, IOS Press, Amsterdam, 1999, pp. 251–292.
[84] I. Stakgold, Green's Functions and Boundary Value Problems, John Wiley & Sons, New York, 1979.
[85] R.P. Stanley, Differentiably finite power series, European J. Combin. 1 (2) (1980) 175–188.

[86] G. Sutcliffe, C. Suttner, The TPTP problem library: CNF Release v1.2.1, J. Automat. Reason. 21 (2) (1998) 177–203.
[87] T. Tammet, Gandalf, J. Automat. Reason. 18 (2) (1997) 199–204.
[88] A. Trybulec, H.A. Blair, Computer aider reasoning, in: R. Parikh (Ed.), Proc. of the Conf. Logic of Programs, in: Lecture Notes in Comput. Sci., vol. 193, Springer, Berlin, 1985, pp. 406–412.
[89] D.H.D. Warren, Higher-order extensions of Prolog: are they needed?, in: Machine Intelligence, vol. 10, Edinburgh University Press, Edinburgh, UK, 1982, pp. 441–454.
[90] C. Weidenbach, B. Afshordel, U. Brahm, C. Cohrs, T. Engel, E. Keen, C. Theobalt, D. Topic, System abstract: Spass version 1.0.0. in: [35], pp. 378–382.
[91] F. Wiedijk, Comparing mathematical provers, in: A. Asperti, B. Buchberger, J.H. Davenport (Eds.), Proc. of the Second Int. Conf. on Mathematical Knowledge Management, Bertinoro, Italy, Springer, Berlin, 2003, pp. 188–202.
[92] W. Windsteiger, A set theory prover in Theorema: Implementation and practical applications, PhD thesis, RISC, Johannes Kepler University, Linz, Austria, May 2001.
[93] S. Wolfram, The Mathematica Book, fifth ed., Wolfram Media Inc., 2003.
[94] W.T. Wu, Basic principles of mechanical theorem proving in elementary geometries, J. Automat. Reason. 2 (1986) 221–252.

Theorema

Wolfgang Windsteiger, Bruno Buchberger, and Markus Rozenkranz

Formalization and answers by Wolfgang Windsteiger <Wolfgang.Windsteiger@risc.uni-linz.ac.at>, Bruno Buchberger <Bruno.Buchberger@risc.uni-linz.ac.at> and Markus Rosenkranz <Markus.Rosenkranz@risc.uni-linz.ac.at>.

12.1 Statement

Theorem[“sqrt[2] irrational”, $\neg$ rat[$\sqrt{2}$]]

12.2 Definitions

Definition[“rational”, any[r],
rat[r] $:\Leftrightarrow \underset{a,b}{\exists}$ (nat[a] $\wedge$ nat[b] $\wedge$ $r = \frac{a}{b} \wedge$ coprime[a, b])]

Definition[“sqrt”, any[x],
$\sqrt{x} := \underset{y}{\ni!}$ ($y^2 = x$)]

12.3 Proof

Load Theorema and Set Preferences

Needs[“Theorema‘”]

SetOptions[Prove, transformBy → ProofSimplifier,
TransformerOptions → {brances → Proved, steps → Essential}];

The Proof of the Main Theorem

Theorem[“sqrt[2] irrational”, $\neg$ rat[$\sqrt{2}$]]

Definition[“rational”, any[r],
rat[r] $:\Leftrightarrow \underset{a,b}{\exists}$ (nat[a] $\wedge$ nat[b] $\wedge$ $r = \frac{a}{b} \wedge$ coprime[a, b])]

Definition[“sqrt”, any[x],
$\sqrt{x} := \underset{y}{\ni!}$ ($y^2 = x$)]

Lemma[“coprime”, any[a, b], with[nat[a] $\wedge$ nat[b]],
($2b^2 = a^2$) $\Rightarrow \neg$ coprime[a, b]]

F. Wiedijk (Ed.): The Seventeen Provers of the World, LNAI 3600, pp. 96–107, 2006.

```
Prove[Theorem["sqrt[2] irrational"],
    using → ⟨Lemma["coprime"], Definition["rational"], Definition["sqrt"]⟩,
    built-in → Built-in["Rational Numbers"], by → ElementaryReasoner,
    ProverOptions → {SimplifyFormula → True, RWCombine → True}];
```

Prove:

(Theorem (sqrt[2] irrational)) $\neg\, \mathrm{rat}[\sqrt{2}]$,

under the assumptions:

(Lemma (coprime)) $\underset{a,b}{\forall}\, (\mathrm{nat}[a] \wedge \mathrm{nat}[b] \Rightarrow ((2 * b^2 = a^2) \Rightarrow \neg\, \mathrm{coprime}[a, b]))$,

(Definition (rational)) $\underset{r}{\forall}\, \mathrm{rat}[r] :\Leftrightarrow \underset{a,b}{\exists}\, ((\mathrm{nat}[a] \wedge \mathrm{nat}[b]) \wedge (r = \frac{a}{b} \wedge \mathrm{coprime}[a, b]))$,

(Definition (sqrt)) $\underset{x}{\forall}\, \sqrt{x} := \underset{y}{\exists!}\, y^2 = x.$

From what we already know follows:
From (Definition (sqrt)) we can infer by expansion of the "such that"-quantifier

(1) $\underset{x,y}{\forall}\, (\sqrt{x} = y \Leftrightarrow y^2 = x).$

We prove (Theorem (sqrt[2] irrational)) by contradiction.
We assume

(3) $\mathrm{rat}[\sqrt{2}]$,

and show a contradiction.
Formula (3), by (Definition (rational)), implies:

(4) $\underset{a,b}{\exists}\, (\mathrm{coprime}[a, b] \wedge \mathrm{nat}[a] \wedge \mathrm{nat}[b] \wedge \sqrt{2} = \frac{a}{b})$.

By (4) we can take appropriate values such that:

(5) $\mathrm{coprime}[\mathrm{a}_0, \mathrm{b}_0] \wedge \mathrm{nat}[\mathrm{a}_0] \wedge \mathrm{nat}[\mathrm{b}_0] \wedge \sqrt{2} = \frac{\mathrm{a}_0}{\mathrm{b}_0}$.

By modus ponens, from (5.2), (5.3) and an appropriate instance of (Lemma (coprime)) follows:

(6) $2 * {\mathrm{b}_0}^2 = {\mathrm{a}_0}^2 \Rightarrow \neg\, \mathrm{coprime}[\mathrm{a}_0, \mathrm{b}_0]$,

Formula (5.4), by (1), implies:

(7) $\left(\frac{\mathrm{a}_0}{\mathrm{b}_0}\right)^2 = 2.$

Using built-in simplification rules we can simplify the knowledge base: Formula (7) simplifies to

(8) $2 * {b_0}^2 = {a_0}^2$.

From (8) and (6) we obtain by modus ponens

(9) $\neg$ coprime$[a_0, b_0]$.

Now, (9) and (5.1) are contradictory.

The Proof of the Auxiliary Lemma

The auxiliary Lemma "coprime" is a statement essentially about natural numbers. In the spirit of theory exploration we assume this lemma to be proven during an (earlier) exploration of the notion "coprime" within the universe of natural numbers. In this section, we show this phase of exploration of the natural numbers.

Lemma["coprime", any$[a, b]$,
$(2b^2 = a^2) \Rightarrow \neg$ coprime$[a, b]$]

Definition["even", any$[a]$,
is-even$[a] :\Leftrightarrow \underset{m}{\exists} (a = 2m)$]

Proposition["even numbers", any$[a, b]$,
$(2b = a) \Rightarrow$ is-even$[a]$ "characteristic"
is-even$[a^2] \Rightarrow$ is-even$[a]$ "even square"]

Proposition["common factor", any$[a, b]$,
$\neg$ coprime$[2a, 2b]$]

Prove[Lemma["coprime"], using $\rightarrow$ ⟨Proposition["even numbers"],
Proposition["common factor"], Definition["even"]⟩,
built-in $\rightarrow$ Built-in["Natural Numbers"], by $\rightarrow$ ElementaryReasoner,
ProverOptions $\rightarrow$ {SimplifyFormula $\rightarrow$ True}, SearchDepth $\rightarrow$ 40];

Prove:

(Lemma (coprime)) $\underset{a,b}{\forall} ((2 * b^2 = a^2) \Rightarrow \neg$ coprime$[a, b])$,

under the assumptions:

(Proposition (even numbers): characteristic) $\underset{a,b}{\forall} ((2 * b = a) \Rightarrow$ is-even$[a])$,

(Proposition (even numbers): even square) $\underset{a}{\forall}$ (is-even$[a^2] \Rightarrow$ is-even$[a])$,

(Proposition (common factor)) $\underset{a,b}{\forall} \neg \text{coprime}[2 * a, 2 * b]$,

(Definition (even)) $\underset{a}{\forall} \text{is-even}[a] :\Leftrightarrow \underset{m}{\exists} (a = 2 * m)$.

We assume

(1) $2 * \text{b}_0{}^2 = \text{a}_0{}^2$,

and show

(2) $\neg \text{coprime}[\text{a}_0, \text{b}_0]$.

We prove (2) by contradiction.
We assume

(3) $\text{coprime}[\text{a}_0, \text{b}_0]$,

and show a contradiction.
Formula (1), by (Proposition (even numbers): characteristic), implies:

$\text{is-even}[\text{a}_0{}^2]$,

which, by (Proposition (even numbers): even square), implies:

$\text{is-even}[\text{a}_0]$,

which, by (Definition (even)), implies:

(4) $\underset{m}{\exists} (\text{a}_0 = 2 * m)$.

By (4) we can take appropriate values such that:

(5) $\text{a}_0 = 2 * \text{m}_0$.

Formula (3), by (5), implies:

(6) $\text{coprime}[2 * \text{m}_0, \text{b}_0]$.

Formula (1), by (5), implies:

(7) $2 * \text{b}_0{}^2 = (2 * \text{m}_0)^2$.

Using available computation rules we can simplify the knowledge base:
Formula (7) simplifies to

(8) $2 * \text{m}_0{}^2 = \text{b}_0{}^2$.

Formula (8), by (Proposition (even numbers): characteristic), implies:

$\text{is-even}[\text{b}_0{}^2]$,

which, by (Proposition (even numbers): even square), implies:

$\text{is-even}[\text{b}_0]$,

which, by (Definition (even)), implies:

(9) $\underset{m}{\exists}\,(\mathrm{b}_0 = 2 * m)$.

By (9) we can take appropriate values such that:

(10) $\mathrm{b}_0 = 2 * \mathrm{m}_1$.

Formula (6), by (10), implies:

(15) $\mathrm{coprime}[2 * \mathrm{m}_0, 2 * \mathrm{m}_1]$.

Now, (15) and (Proposition (common factor)) are contradictory.

12.4 System

What is the home page of the system?

<http://www.theorema.org/>

What are the books about the system? The essential theoretical ideas involved in the *Theorema* system can be found already in [5]. A comprehensive description of the basic design of a computer-system based on these ideas is then given in [1]. For a more elaborate exposition and for concrete design principles of the current *Theorema* system, we refer to the survey papers [[3,4]].

Most of the system components are described in all detail in journal papers, conference proceedings articles, or technical reports. All downloadable material can be found on the *Theorema* homepage.

What is the logic of the system? The logic frame of *Theorema* is higher order predicate logic, which is extended by the language construct "sequence variables". Sequence variables are variables for which an arbitrary finite number of terms can be substituted. Sequence variables are a convenient construct for the formulation of algorithms in terms of pattern matching within logic. Thus, the *Theorema* language is also particularly suited for expressing logic algorithms like theorem provers etc. A logical study of sequence variables is given in [6].

In fact, the *Theorema* system is a (growing) collection of various general purpose and special theorem provers. The general purpose provers (like the first order predicate logic natural deduction prover) are valid only for special fragments of predicate logic (e.g. first order predicate logic). The special provers are valid only under the additional assumption that special knowledge is available that characterizes the underlying special theory. For example, the (various versions) of the induction prover assume that, for certain functions, an induction principle holds; the geometry prover based on the Gröbner bases method assumes that the universe of discourse is the field of complex numbers and the basic properties of complex numbers are available; the set theory prover assumes that the axioms and basic properties of set theory are valid. We do not yet have

a prover for a general version of higher order logic. Of course, this approach to building a theorem proving system supposes that the correctness of special theorem provers is (automatically) proved w.r.t. to more general provers. So far, this research program is only partially carried out. For example, Gröbner bases theory is proved correct in a fairly formal way. A complete formal proof within *Theorema* is planned for the near future.

What is the implementation architecture of the system? All *Theorema* 'reasoners' (provers, solvers, and simplifiers) are written in the programming language of Mathematica. *Theorema* does *not* use the Mathematica algorithm library or any implicit mathematical knowledge presupposed in Mathematica algorithms. The *Theorema* language is a version of higher order predicate logic. The currently available reasoners comprise *general purpose reasoners* (several methods for proving in first order predicate logic, a prover for equational reasoning, or a general simplification prover based on equality rewriting) and special purpose reasoners (like Collins' decision procedure for the theory of real closed fields, a special prover for geometry based on algebraic techniques like Gröbner bases, or the simplification prover for number domains used in the proofs shown in Section 12.3) that are valid only in particular theories. Various external provers like e.g. Otter can be accessed through *Theorema* by translating *Theorema* formulae to the specific input format for these provers and getting the results back. Reasoners available in Mathematica can also be accessed by *Theorema* upon explicit request by the user. The *Theorema* language includes a programming language as a natural sub-language.

Theorema provers have a modular structure, i.e. every *Theorema* prover is composed from smaller units, so-called 'special prover modules'. These prover modules are separate units, and can therefore be combined in arbitrary way. In the current status, the access to special prover modules is restricted to the system developers, but a mechanism for users to compose their own provers from available special prover modules is planned for future versions of the system. *Theorema* uses a general proof search procedure that maintains a global proof object. The proof object has a tree-structure where each node in the tree represents a *proof situation* made up basically from the proof goal and the current assumptions. In each step, the proof search procedure tries to apply a special prover module in order to simplify the current proof situation. A special prover module can be applied to a proof situation if one of its inference rules can be applied to the proof situation. Inference rules in the prover modules are implemented as Mathematica programs that take a proof situation as parameter and return a new proof situation. Applicability of an inference rule is tested by pattern matching on the parameters of the Mathematica program against the current proof situation. When applying a special prover module to a proof situation the proof search procedure inserts the result of the first applicable rule as a new node into the global proof object. The proof search continues until a trivial proof situation (e.g. the goal is identical to one of the assumptions) appears on one branch of the tree, in which case the proof succeeds, or until the maximal search depth is exceeded on all branches, in which case the proof attempt fails.

What does working with the system look like? The current version of *Theorema* is implemented as an extension package to Mathematica, thus, the standard way of working with *Theorema* is an interactive user-system-dialog in the well-known Mathematica notebook FrontEnd. The Mathematica notebook FrontEnd supports configurable two-dimensional mathematical notation both in input and output. The examples in Section 12.3 demonstrate a typical *Theorema* session. Two categories of commands are available for the user:

Organization of Knowledge: The *Theorema* Formal Text Language allows the user to enter arbitrary definitions, axioms, propositions, algorithms, etc. to the system and combine such formulae into theories in a nested way so that hierarchies of mathematical knowledge bases (theories) can be built up. All formulae may receive key words and labels for easy reference. Labels, however, carry no logical meaning. Declaration of free variables and conditions on these variables are specified using the keywords 'any' and 'with'. The individual formulae can be entered in a two-dimensional syntax very close to how formulae are written in mathematical textbooks. The actual input of arbitrary two-dimensional notation and special mathematical symbols is supported by the standard Mathematica notebook FrontEnd through input palettes and keyboard shortcuts. The input of *Theorema*-specific notation, like e.g. a formal text entity containing labelled formulae as used in Proposition "even numbers", is taken care of by additional input palettes. The definitions, theorems, lemmata, and propositions as they display in Section 12.3 illustrate some of the features of the Formal Text Language.
Note that the *Theorema* language also supports several notational variants for mathematical syntax in order to accommodate to the preferences of the user. As an example, we used the standard form of the existential quantifier

$$\exists_{a,b} \left(\mathrm{nat}[a] \wedge \mathrm{nat}[b] \wedge r = \frac{a}{b} \wedge \mathrm{coprime}[a, b]\right)$$

in Definition "rational" as given in Section 12.2. Supported notational variants, which would all be interpreted by all *Theorema* provers in exactly the same way, are:

$$\exists_{\mathrm{nat}[a],\mathrm{nat}[b]} \left(r = \frac{a}{b} \wedge \mathrm{coprime}[a, b]\right),$$

$$\exists_{\mathrm{nat}[a,b]} \left(r = \frac{a}{b} \wedge \mathrm{coprime}[a, b]\right), \text{ or}$$

$$\exists_{\mathrm{nat}[a] \wedge \mathrm{nat}[b]}^{a,b} \left(r = \frac{a}{b} \wedge \mathrm{coprime}[a, b]\right).$$

Mathematical Activities: Mathematical knowledge can then be processed in various ways using the *Theorema* User Language. Currently, the User Language supports 'proving', 'solving', and 'simplifying', see the commands `Prove[...]` in Section 12.3 for typical examples. These examples also exhibit, how knowledge specified in the Formal Text Language can be referred to in the User Language. Note also, that the keywords such as 'any' or 'with'

are processed when formulae are passed to a prover: Compare the formulae as they are echoed at the beginning of the proof to how they appear in the Formal Text Language.
Proofs (or traces of solving or simplifying) are generated completely automatically and display very much in the form as they would be written in mathematical textbooks including intermediate explanatory text in natural language. Alternatively, *Theorema* also offers an interactive mode, where user-interaction *during proof generation* is supported. Mathematical formulae are displayed in two-dimensional syntax. The proofs appear in the *Theorema* system almost exactly as they are typeset in Section 12.3.
However, some of the features of the *Theorema* user interface cannot be modelled in the style of this paper:

- Proof branches are organized in hierarchically nested cells, which can be opened or closed by double-clicking the cell bracket on the right margin of the window. This allows the user to hide (or display) certain parts of a proof easily by mouse-click.
- Formula labels are active elements, such that clicking a formula reference in the running text shows the full formula in a separate pop-up window.
- Goal formulae, assumptions, labels, and explanatory text use different colors.

Also, there are means in *Theorema* to design and implement one's own syntax including the design of new mathematical symbols of arbitrary complexity ('logicographic symbols').

What is special about the system compared to other systems?

- *Theorema* is both a logic language and a programming language. This means that, for example, within the same language and system, a formula that describes an algorithm can be proved correct and can then be executed on concrete input.
- The three fundamental mathematical activities proving, solving, and simplifying can be done in one uniform language and logic frame.
- *Theorema* is a multi-method system: instead of using one uniform proof method for all of mathematics *Theorema* provides sophisticated special provers for certain mathematical theories. These special provers are partly based on powerful computer algebra methods, which were the research focus of the working group in earlier years.
- *Theorema* has an attractive two-dimensional, extensible syntax.
- Most of the *Theorema* provers generate proofs in a natural, human-readable style with intermediate explanatory text and various tools that help to get various (contracted and expanded) views of proofs.
- *Theorema* has various structuring mechanisms for large mathematical knowledge bases, notably the recursive 'Theory' construct and a functor construct, which is similar to but more general than the functor construct of SML.

What are other versions of the system? There are no other versions of the system. The current version is free for download from the *Theorema* webpage under "software".

Who are the people behind the system? The development of *Theorema* has been initiated by Bruno Buchberger, who also implemented first prototypes in the mid 1990's and directs the project since then in cooperation with Tudor Jebelean and Wolfgang Windsteiger. Current senior *Theorema* researchers are, in addition, Temur Kutsia, Florina Piroi, and Markus Rosenkranz. Former *Theorema* PhD students, who contributed to the system, are Daniela Vasaru-Dupre, Mircea Marin, Koji Nakagawa, Judit Robu, and Elena Tomuta. For a complete (and always up-to-date) listing of persons involved in the *Theorema* project we refer to the *Theorema* webpage.

What are the main user communities of the system? There is a small community of alpha testers, mainly math researchers and math teachers. A major didactic case study in undergraduate math education at the Johannes Kepler University of Linz and the Polytechnic University Hagenberg is under way.

What large mathematical formalizations have been done in the system? Elementary analysis (with the typical epsilon/delta proofs), equivalence relations and partitions based on set theory, polynomial interpolation, the theory of lists with verified list algorithms like sorting, and the automated synthesis of the Gröbner bases algorithm.

What representation of the formalization has been put in this paper? The formalization in Section 12.3 should be self-explanatory. In fact, it is a main design principle of *Theorema* that formalizations using the *Theorema* language constructs should be self-explanatory and easy to read. The theorem is formulated as a statement over the positive real numbers. The first part shown in Section 12.3 contains the main proof that reduces the problem over the positive reals to a lemma over the natural numbers. Section 12.3 shows the second part that contains the proof of the auxiliary lemma over the naturals.

What needs to be explained about this specific proof? In the proof of the main theorem, we assume that all variables range over the *positive real numbers*, i.e. all formulae in the knowledge base should be true statements in the domain of positive real numbers.

The proof of the theorem as shown in Section 12.3 is generated completely automatically within *Theorema*. The only user-interactions required are

- to choose an appropriate prover from the *Theorema* prover library (in the example the "Elementary Reasoner"),
- to specify appropriate prover options for the chosen prover, and
- to provide the auxiliary Lemma "coprime" necessary in the proof.

In the spirit of the layered approach of *Theorema*, the auxiliary lemma can then be proved in a separate proving session as shown in Section 12.3. Again, auxiliary knowledge needed in this proof (in the example the Propositions "even numbers" and "common factor") can be proved in separate phases of exploring a theory.

However, we do not present the proofs of these propositions here, because when investigating the irrationality of $\sqrt{2}$ one would usually consider these propositions to be *known properties of natural numbers*. Phases of theory exploration can be structured bottom-up or top-down just like human mathematicians build up hierarchically structured mathematical knowledge.

Typically, different phases of theory exploration are characterized by using different proving techniques. In the case of *automated theory exploration* using *Theorema* this means that switching from one exploration phase to another would be reflected in changing the prover that is used in the Prove-calls. In this example, in fact, we use the Elementary Reasoner in both phases, but we allow the prover to access different portions of built-in computational knowledge in either phase.

The special feature of the Elementary Reasoner used for generating the proofs shown in Section 12.3 is the smooth integration of 'proving' and 'simplifying' ('computing') within one system. 'Simplifying', here, means 'simplifying expressions (to canonical form) based on the *algorithmic semantics* of the language'. The algorithmic semantics of the *Theorema* language consists of computation rules for the *algorithmic part of the language*[1], i.e. finite sets, finite tuples, quantifiers with a finite range, and basic arithmetic on numbers. Sets, tuples, and quantifiers are represented as special data structures and the operations on these entities are implemented in the *Theorema* language semantics using the programming language of Mathematica. Only for arithmetic on natural numbers, integers, and rational numbers the *Theorema* semantics may access the arithmetic rules from the underlying Mathematica system.

In the example, the option `built-in→Built-in["Rational Numbers"]` allows the Elementary Reasoner explicitly to use built-in computation rules for operations on rational numbers. Arithmetic on numbers is, in fact, the only case, where *Theorema* silently relies on the mathematical algorithm library available from Mathematica[2]. On explicit user request, however, the interface allows this prover to even access special simplification algorithms from Mathematica when performing computational simplification. Specifying the prover option `SimplifyFormula→True` (default value is `False`) tells the prover to post-process any formula obtained from a computation by Mathematica's `FullSimplify` function. `FullSimplify` is a black-box simplifier for Mathematica expressions, which uses powerful simplification rules, in particular for arithmetic expressions. Moreover, the prover performs additional simplification of equalities involving arithmetic expressions, such that certain equalities are turned into equalities that are more likely to be usable for rewriting. In our context, the correctness of the built-in simplifier is based exclusively on the field axioms (for proofs over the reals) and the ring axioms (for proofs over the naturals[3]), respectively. In

[1] In contrast, the semantics of the *non-algorithmic part of the language* is coded into inference rules that make up the *Theorema* special prover modules.

[2] Apart from that, Mathematica is used just as a programming environment!

[3] To be precise: the simplifier for the naturals uses the ring axioms without the axiom ensuring the existence of additive inverses.

other words, the 'hidden knowledge' used by the simplifier are only the field or ring axioms[3], respectively. All other knowledge, e.g. Lemma "coprime", Proposition "even numbers", and Proposition "common factor", must be mentioned explicitly in the call of the prover.

A combination of these capabilities is used in the main proof when simplifying formula

$$(7) \quad \left(\frac{\mathrm{a}_0}{\mathrm{b}_0}\right)^2 = 2$$

to

$$(8) \quad 2{\mathrm{b}_0}^2 = {\mathrm{a}_0}^2$$

and in the proof of the auxiliary lemma when simplifying

$$(7) \quad 2{\mathrm{b}_0}^2 = (2\mathrm{m}_0)^2$$

to

$$(8) \quad 2{\mathrm{m}_0}^2 = {\mathrm{b}_0}^2.$$

Definition "sqrt" uses a convenient language construct available in *Theorema*: The 'the-unique'-quantifier $\underset{y}{\ni!}\ P_y$ denotes 'the unique y satisfying P_y'. The expression $f[x] := \underset{y}{\ni!}\ P_{x,y}$ is a means to express an implicit definition for the function f, for which, of course, we need to verify $\underset{x}{\forall}\ \underset{y}{\exists!}\ P_{x,y}$ beforehand. In the concrete example of Definition "sqrt", we assume that $\underset{x}{\forall}\ \underset{y}{\exists!}\ y^2 = x$ holds over the positive real numbers. Based on this convention the prover applies an inference rule for the 'the-unique'-quantifier in order to rewrite Definition "sqrt" in the main proof into

$$(2) \quad \underset{x,y}{\forall}\ (\sqrt{x} = y \Leftrightarrow y^2 = x).$$

All other steps in the proofs are basic predicate logic and therefore require no further explanation.

References

1. B. Buchberger. Symbolic Computation: Computer Algebra and Logic. In K. Schulz, editor, *FroCoS: Frontiers of Combined Systems*, 1996.
2. B. Buchberger. Algorithm Supported Mathematical Theory Exploration. In B. Buchberger and John Campbell, editors, *Proceedings of AISC 2004 (7th International Conference on Artificial Intelligence and Symbolic Computation*, volume 3249 of *Springer Lecture Notes in Artificial Intelligence*, RISC, Johannes Kepler University, Austria, September 22-24 2004. Springer Berlin Heidelberg. ISSN 0302-9743, ISBN 3-540-23212-5.

3. B. Buchberger, C. Dupre, T. Jebelean, F. Kriftner, K. Nakagawa, D. Vasaru, and W. Windsteiger. The Theorema Project: A Progress Report. In M. Kerber and M. Kohlhase, editors, *Symbolic Computation and Automated Reasoning (Proceedings of CALCULEMUS 2000, Symposium on the Integration of Symbolic Computation and Mechanized Reasoning)*, pages 98–113. St. Andrews, Scotland, Copyright: A.K. Peters, Natick, Massachusetts, 6-7 August 2000.
4. B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuta, and D. Vasaru. A Survey of the Theorema project. In W. Kuechlin, editor, *Proceedings of ISSAC'97 (International Symposium on Symbolic and Algebraic Computation, Maui, Hawaii, July 21-23, 1997), ACM Press 1997*, pages 384–391, 1997.
5. B. Buchberger and F. Lichtenberger. *Mathematik für Informatiker I.* Springer Verlag, 2nd edition, 1981.
6. T. Kutsia and B. Buchberger. Predicate Logic with Sequence Variables and Sequence Function Symbols. In Andrea Asperti, Grzegorz Bancerek, and Andrzej Trybulec, editors, *Proceedings of the 3rd International Conference on Mathematical Knowledge Management, MKM'04*, volume 3119 of *Lecture Notes in Computer Science*, pages 205–219, Bialowieza, Poland, Sep 19–21 2004. Springer Verlag.

A. Bigatti, S. Ranise (eds.), Proceedings of Calculemus 2006, pp. 35–50

Analytica V: Towards the Mordell-Weil Theorem

Edmund M. Clarke [a,1] Avi S. Gavlovski [a,2]
Klaus Sutner [a,3] and Wolfgang Windsteiger [a,b,4]

[a] *Dept. of Computer Science, Carnegie Mellon University, USA*
[b] *RISC Institute, JKU Linz, Austria*

Abstract

Analytica V is a theorem proving system that is built on top of the symbolic computation system *Mathematica*. It was originally designed by E. Clarke and X. Zhao in the early 1990's. We describe here a redesign of the system that extends its abilities to reasoning about some aspects of number theory.

1 Introduction

Analytica was originally designed to reason about 19^{th} century mathematics, in particular elementary calculus and number theory. The system was developed in the early 1990's by Xudong Zhao and Edmund Clarke, see [CZ92,BCZ98], an intermediate version of the system has been described in [CKOS03]. The system relies heavily on symbolic computation performed by *Mathematica*, the underlying computer algebra system, see [Wol02]. Apart from built-in simplification and decision procedures, Analytica also uses internal rewrite rules as well as decision procedures. For example, to deal with linear inequalities over the reals it can either exploit its own implementation of the standard elimination procedure or *Mathematica*'s linear programming algorithm. For non-linear inequalities we have implemented a variant of Bledsoe's sup-inf method.

In this paper we report on Analytica V, a redesign of the system that extends its capabilities to reason about certain concepts in number theory. In contrast to [CKOS03], where the emphasis was put on methods to connect

[1] Email: emc@cs.cmu.edu
[2] Email: agavlovs@andrew.cmu.edu
[3] Email: sutner@aiki.ccaps.cs.cmu.edu
[4] Email: Wolfgang.Windsteiger@risc.uni-linz.ac.at

Analytica to an external mathematical knowledge data base (MBase), this paper describes recent enhancements in the internal system design and the reasoning engine. As a mid-range goal, we consider the automated generation of proofs of the Mordell-Weil Theorem (the group of rational points on an elliptic curve is finitely generated) and the Dirichlet Theorem (the infinitude of primes in arithmetic progressions), see for example [Sil86]. All examples presented in this work relate to the formalization of these theorems.

As its predecessor system, Analytica V focuses on proofs whose logical complexity can be reduced greatly by algebraic simplification and symbolic computation, or by application of various decision procedures. For example, the proof of associtivity of the group operation on an elliptic curve relies heavily on Gröbner basis computations.

The key features of Analytica V that we want to describe in this work are:

- the setup of inference rules and the general proof-search mechanism,
- a look-up method for efficient check of side-conditions in computations, and
- some specialized techniques developed specifically in connection with the Mordell-Weil theorem.

We begin with a description of the Analytica V language and its general proof search mechanism in Section 2; the look-up mechanism is explained in Section 3. A number of specialized techniques that are relevant to the proofs shown in Section 5 are detailed in Section 4.

2 Language, Inference Rules, and Proof Search

2.1 The Analytica V Language

The object-language in the Analytica system is essentially first-order predicate logic, though we allow for restricted higher-order concepts. Notably, we allow *currying* and *universally quantified function and predicate variables in the goal*. The idea is, however, to eliminate curryed expressions by expansion of definitions given for the curryed function or predicate and to allow parameters to stand for functions and predicates. Proving goals containing existentially quantified function or predicate variables, which would require "solving for functions and predicates" in some form is not our current aim. Rather, we use currying and higher-order variables to give natural representations of mathematical concepts, notably in our examples in group-theory. The example in Section 5.1 shows typical applications of currying when dealing with group characters and homomorphisms.

Many of the terms and atomic predicates in the language carry *domain information*. Thus, we represent multiplication on the natural numbers by a term [5] `times[NN,a,b]`, whereas multiplication in an arbitrary group G is

[5] We will use *Mathematica*-syntax or *Mathematica*-oriented pseudo code when describing data structures or program fragments of Analytica V. In some cases, we will refer to internal

given by a term `times[G,a,b]`. In a similar vein, `eq[QQ,a,b]` denotes equality over the rationals, and so on. We have found easy access to additional domain information to be very helpful in the application of simplifiers and various decision procedures. It also helps somewhat in improving the legibility of Analytica-generated formulae.

This approach is similar, although implemented in slightly different style, to the use of functors in the Theorema system, see [Win99]. There is a complete separation between the Analytica object-level language and the *Mathematica* programming language—an aspect, in which Analytica *V* differs from earlier versions of Analytica. In order to have access to simplification and decision procedures provided by *Mathematica* we have a translator between the two languages.

2.2 Proofs, Proof States, and Inference Rules

The proof state always consists of a list of sequents of the form

$$\texttt{seq[\{}l_1,\ldots,l_a\texttt{\},\{}r_1,\ldots,r_c\texttt{\}]},$$

where the l_i and r_i represent the (conjunction of) assumptions and the (disjunction of) conclusions, resprectively. The proof starts with just the initial sequent `seq[{},{`g`}]`, where g is the goal formula to be proven. As always, a successful proof is a sequence of applications of *proof rules* that leads to an empty proof state.

All Analytica inference rules are checked into the system as pairs (T, R) where T is an applicability test and R implements the actual transformation on the proof state. T typically returns a (possibly empty) list of sequents as output. The applicability test relies heavily on pattern matching in *Mathematica*.

Proof search will be described in detail in Section 2.3. The key idea is to think of inference rules "listening for events", where by "event" we mean formulae appearing new in the current sequent. We will call the current sequent *together* with the current event our *current proof situation*. In order to relieve the individual inference rules of having to determine the actual event, the proof search will decide applicability of an inference rule by calling the associated applicability test on the current proof situation. The test can perform arbitrary tests on the proof situation, in many cases, however, pattern matching is sufficient. For efficiency reasons we provide a mechanism that allows expressions computed during the applicability test to be passed to the actual rule application in order to avoid recomputation. Therefore, any list $\{i_1,\ldots,i_t\}$ as the return value of an applicability test is interpreted as "the rule is applicable at positions $i_1,\ldots,i_t$", any other return value is interpreted as "the rule does not apply". Consequently, the implementation of an inference rule takes $i_1,\ldots,i_t$ as parameters in addition to the current sequent.

symbols used in the implementation, such as '`NN`' in this case for 'the natural numbers'.

```
While[ ProofState != EMPTY,
    EnqueueRules[
        QueryRulebase[CurrentSequent[],CurrentEvent[]]];
    If[ EmptyRuleQueue[],
        DetectOutOfRules[],
        ApplyProofRule[DequeueRule[]];
    ]]
```

Fig. 1. The Analytica V proof search procedure.

As we will see later, it is convenient to implement an inference rule `R` using currying as `R[`$i_1, \ldots, i_t$`][seq[l,r]]`.

2.3 The Proof Search and the Prover Configuration

Analytica does not provide *one prover*. Rather, it provides a *generic proof search procedure*, which employs a global database of inference rules, the *rule base*. Arbitrary provers can be assembled by composing a rule base, i.e. a list of names of inference rules, for the proof search procedure. Apart from the rule base, the *prover configuration* consists of several global system settings that define the knowledge base relative to which a certain proof is carried out, such as definitions of predicates and functions or basic properties that are assumed to hold for a particular proof, see also Section 3.2.1 and the example in Section 5.1. The prover configuration can be adjusted for each proof individually and, in some sense, it should be seen as (part of) the "underlying theory" for the proof.

The top-level proof search itself is then rather simple, the pseudo-code given in Figure 1 is nearly identical to the actual *Mathematica* code and should almost be self-explanatory. At its center, it *queues inference rules* found by `QueryRulebase`, which simply tests all rules in the rule base using their applicability test on the current sequent with the current event. Assume the rule `R`'s applicability test resulted in $\{i_1, \ldots, i_t\}$, then `QueryRulebase` will actually return `R[`$i_1, \ldots, i_t$`]` to be stored in the rule queue. When dequeuing the rule from the queue later, it will therefore have the information passed from the rule test attached to it. `ApplyProofRule` will then apply the entire expression to the current sequent, thus resulting in `R[`$i_1, \ldots, i_t$`][seq[l,r]]` as the actual call of the inference rule `R`.

Note that we store *all rules* applicable to the current proof situation in the rule queue although we apply then only the first one. This means that, when we enqueue new rules, there might be still rules from previous cycles in the queue. We do not simply queue the new ones at the end, but we always sort the queue w.r.t. the age of a rule—a notion that takes into account the last time when a rule was applied. This strategy gives priority to the new rules queried but it prevents older rules from "starving in the queue".

Once we are faced with an empty queue, i.e. no new rules apply to the current proof situation and there are no more pending rules applicable to previous situations, we detect that we essentially "ran out of ideas". The

proof does not necessarily fail at this point; rather, we increase our "level of despair", and only if that level reaches a certain threshold the proof attempt fails. We will provide certain inference rules that make their applicability test depend on the current level of despair. Hence, by gradually increasing the level of despair, an inference rule may become applicable at some point although it did not apply to the same proof situation earlier on. This seems to conform well with the approach taken by a human prover: As long as standard techniques apply, one uses those to make progress in the proof. Once one gets stuck, one considers more complicated (and, in our case, computationally more expensive) methods or methods that may not seem immediately promising. In the current implementation it is advisable to guard all inference rules that involve complicated and potentially time-consuming symbolic computations in this fashion against eager application.

In addition, there are default actions that can be performed before as well as after the rule taken from the queue. In the current status, the only rule applied before the queued rules is a normalization rule, which takes care about the propositional connectives and the logical quantifiers. For example, normalization will perform And-Splits in the antecedent.

The introduction of meta-variables[6] for existentially quantified variables in the consequent happens, mainly due to administrative reasons, in a separate rule. Note also, that case splits for disjunctions in the antecedent (Or-Left) and conjunctions in the consequent (And-Right) are not performed during normalization. We implemented separate rules for Or-Left and And-Right, because we want to split the sequent only if we are out of other rules. This is important, in particular, for instantiation of evars. Since we will try to instantiate evars in conjunctions in certain situations by symbolic computation decision procedures, e.g. algorithms for finding solutions of systems of algebraic equations or inequalities, it is beneficial to not perform the And-Split immediately.

The maintenance of the proof state and the current event, the actual application of an inference rule to the current sequent, and the maintenance of the individual sequents is accomplished by `ApplyProofRule` and its companion programs. Most importantly, we perform certain simplifications on new sequents before they enter the proof state. These simplifications can be

sequent-level simplifications, such as removing duplicate formulae, occurrences of `true`/`false` etc. from the antecedent/consequent, closing a sequent if `true`/`false` appear in the consequent/antecedent, removing formulae in the consequent that "follow easily" from the antecedent (we refer to Section 3 for when we consider a formula to "follow easily"),

formula-level simplifications, which are mainly boolean simplifications on

[6] We sometimes refer to a meta-variable as an "evar" (for "existential variable"). Introducing meta-variables for existential goals is a well-known technique used in many systems nowadays.

individual formulae, such as handling double-negation, removing duplicate subformulae in conjunctions/disjunctions and the like, or

custom simplifications, such as simplifying equalities/inequalities/disequalities with identical subterms. Additional custom simplifications to be applied at this stage—very convenient for user-defined functions or predicates—can be specified through *Mathematica* transformation rules stored in a global variable.

Having these simplifications at a central place is very helpful, since the individual inference rules can then neglect this aspect entirely, i.e. they can just compose new formulae and their simplification will be taken care of automatically.

3 Backward-Reasoning for Checking Side-Conditions

3.1 The Problem

The idea of a small, efficient backward-reasoning unit originally arose during the implementation of a simplifier for group-theory but it developed into a system component, which turned out to be applicable in many situations, both *during simplification* and *during logical reasoning*. The problem at hand is one of the central Calculemus-problems, namely the integration of reasoning techniques into computation. As an example, consider a simplification method for expressions in group-like structures to be applied during a proof. Given associativity, we want to re-arrange parentheses; if we have an identity, we want to cancel identity terms and so on. In general, when we need to check a condition C w.r.t. our current knowledge Γ, it may require a proof that C follows from Γ. However, we do not wish to employ a full-blown prover at that point: on the one hand, the argument often does not require a complicated reasoning, on the other hand it may well require a careful choice of reasoning rules—a task handled by the user in our current setting [7] . In real-world proofs, though C may not be contained in Γ, little more than unfolding of definitions is required to derive C from Γ. In the above example, we might know that "we are dealing with an abelian group". Expanding this fact we obtain all the required information. However, it is inefficient to eagerly expand definitions in Γ and apply forward-reasoning techniques for three reasons:

(i) it unnecessarily inflates Γ,

(ii) it is difficult to determine beforehand when to stop expanding, and

(iii) for the human reader of the proof, these steps are (mostly) uninteresting.

[7] Admittedly, using a standard reasoning engine that requires no user-interaction whatsoever, such as, e.g., either of the powerful resolution provers competing in every year's CASC, would be an option.

3.2 The Analytica-Solution

Our approach is to devise a simplified reasoning mechanism for looking up facts in some knowledge base Γ, which is tailored towards the needs in a particular proof setting. The key requirements are efficiency, easy extensibility (by new dependencies in the knowledge base), and easy use in its application (for the Analytica developers). Completeness has intentionally no priority, because this would lead us towards full-blown proving. Rather, we will reason only for a restricted class of goals and we will only use a very limited inference scheme, namely we will be able to look up C in Γ if

- $C \in \Gamma$ or C is known due to built-in knowledge [8], or
- C has the form $C_1 \wedge C_2$ and we can look up both C_1 and C_2 in Γ, or
- C has the form $C_1 \vee C_2$ and we can look up either C_1 or C_2 in Γ, or
- there is a formula ϕ, such that $\phi \Rightarrow C$ and we can look up ϕ in Γ.

This translates directly into a *recursive procedure* for looking up C in Γ, with the last case being worth closer inspection. Which ϕ do we choose and which implications do we employ? The idea is to specify the list of available implications as part of the prover configuration and use these implications to build up a look-up table at the beginning of the proof. Using this look-up table we can then do backward reasoning from C to all possible ϕ. As already said in Section 2.3, all formulae part of the prover configuration are assumed to hold, we do neither require them being axioms nor do we require them being proven before they may be put to the prover configuration.

3.2.1 Building the Look-up Table

We require a list of formulae available for building up the look-up table to be specified as part of the prover configuration. Only universally quantified biconditionals or implications are allowed, since these will allow backward reasoning. As for biconditionals, we need to direct each biconditional into an implication, thereby essentially throwing away half of the information. We have not investigated this case in general; at this point we only process "definitions by biconditionals" of the form [9] $\forall x : p[x] \Leftrightarrow Q$ that introduce a new predicate symbol p, and we use such defintions as $\forall x : p[x] \Rightarrow Q$. As for implications, we split up each implication $\forall x : P \Rightarrow Q_1 \wedge \ldots \wedge Q_n$ into $\forall x : P \Rightarrow Q_1$, $\ldots$, $\forall x : P \Rightarrow Q_n$ and proceed recursively. An implication $\forall x : P \Rightarrow Q$ is only processed further if its right-hand side Q is an *atomic proposition*, otherwise it will not be used for look-up purposes, i.e. the backward reasoning described above will only be done from atomic propositions. Finally, we need to process implications of the form $\forall x : P \Rightarrow Q$, where we assume that Q contains no other free variables than x, but not necessarily all of them. If P

[8] A list of built-in facts can be specified as part of the prover configuration, see the example in Section 5.1

[9] In $\forall x : P$ the x can actually stand for a sequence of variables.

and Q share the same free variables x, then we can reason backwards from *any concrete instance* of Q of the from $Q_{x\to t}$ to $P_{x\to t}$. If some variables y are free in P but not in Q, then we observe that $\forall x, y : P \Rightarrow Q$ is equivalent to $\forall x : (\exists y : P) \Rightarrow Q$ and we can reason backwards from any $Q_{x\to t}$ to $\exists y : P_{x\to t}$. The same considerations regarding free variables have been taken into account in the top-level inference rule for the realization of backchaining using implications, see the proof of the image of δ in section 5.2 for an example of backchaining.

Along these lines, we have implemented a framework that generates the recursive look-up procedure—in form of recursive definitions for a *Mathematica* program `LookupFacts`—automatically from the given list of formulae and from an internal list of "known implications". For looking up $\exists y : P$, we allow a special construct `forsome[y]` to occur in formulae. After a successful look-up we have access to an appropriate binding for y via `this[y]`. This allows for a rather elegant programming style, otherwise not available in *Mathematica*, as shown in the following statement taken from the group simplifier:

```
If[LookupFacts[isIdentity[forsome[n], op, G]],
  id = this[n];
  ... (* cancel op[G,id,_] and op[G,_,id] from expr *)
```

This mechanism is used extensively in our formalization of group theory, it is obviously applicable in other contexts as well.

3.2.2 Where We Use Look-up

As mentioned earlier, the look-up mechanism was originally designed just for checking side-conditions during computations and simplifications, but it quickly turned out that it has a much broader spectrum of applicability. In the current implementation, we exploit `LookupFacts` in the following places:

- In the group simplifiers, see Section 4 for testing the group properties and for actually finding the group operation, the identity, and the inverse,
- In the simplification of sequents, see Section 2.3. In usual sequent calculus, one would delete a formula A in the consequent if A appears in the antecedent. We delete A, if we can look up A in the antecedent! Similarly, we delete parts of conjunctions in the consequent as soon as they can be looked up, and so forth.
- Custom formula simplifications, see Section 2.3, are performed in a context where Γ contains the current antecedent, i.e. any look-up done in a custom simplification rule is relative to the knowledge in the antecedent!
- When translating expressions from Analytica to *Mathematica* we check side-conditions by look-up, e.g. non-zero denominator.

We also refer to the examples in Section 5, which will show the wide range of applicability.

4 Special Techniques

In addition to the logical proof rules, which follow standard sequent calculus, Analytica V contains various specialized rules for special proof situations and/or special theories.

4.1 The Group Simplifiers

We provide a *general-purpose simplifier*, which inspects all terms and simplifies them using associativity, commutativity, identity, and inverses, where checking the properties is done by our look-up mechanism. In addition to that, we also have the possibility to get *specific simplifiers* generated automatically—depending on which properties the structure actually satisfies! As an example, if the knowledge base tells us that G with "+" and "0" forms a monoid, then we can *generate* a simplifier for G that uses associativity of "+" and "0" as the identity w.r.t. "+"—without permanently testing for these properties—and use that simplifier for terms constructed by "+" on G.

4.2 Symmetry Analysis

We analyze conjunctions in the goal and disjunctions in the assumptions to detect symmetries. As a simple example, consider:

$$\texttt{seq}[\{a > 0, b > 0, x > a + b\}, \{x > a \wedge x > b\}]$$

Since the sequent is symmetric about a and b, we can reduce the conjunction in the goal:

$$\texttt{seq}[\{a > 0, b > 0, x > a + b\}, \{x > a\}]$$

More formally, in a sequent $seq[l, r]$, we reduce conjunction $A \wedge B$ in the goal (or disjunction $A \vee B$ in the assumptions), if there exists a permutation f of the parameters in the sequent such that:

$$l[f] =_\alpha l, r[f] =_\alpha r, A[f] =_\alpha B$$

where $=_\alpha$ denotes equality up to alpha conversion and some simple normalizations relating to commutative operators and predicates that are symmetric about their arguments.

We use *Mathematica*'s `Solve`, `Reduce` and `Eliminate` method, see [Wol02], to obtain suggested instantiations of variables in systems of equations over the reals or complexes. In deciding which sets of variables to eliminate when solving a given system, we make use of the symmetry machinery: if the sequent in symmetric about variables a and b, we either try to eliminate both a and b or neither of the two.

4.3 Heuristic Instantiation

In cases where unification and equation solving are insufficient we use an instantiation heuristic. As an example, consider the following sequent:

$$\texttt{seq}[\{y \in \mathbb{Z}, \forall_{x \in \mathbb{Z}} f(x) < f(x+1)\}, \{f(y) < f(y+2)\}]$$

Since $f(y)$ is a term present in the sequent, it is natural to instantiate the universal quantifier on the left with y. To this end we introduce a general rule that considers instantiations of universal quantifiers on the left and existential quantifiers on the right with terms in the sequent, according to built-in heuristics that evaluate the formula resulting from an instantiation.

4.4 Mathematica *Built-Ins*

Needless to say, Analytica$V$ relies heavily on the symbolic capabilities of *Mathematica* for simplification and for decision procedures. The operations used most frequently in the current proofs are the following.

- `FindInstance` is a decision procedure that applies various techniques for systems real, complex, and integer valued equations and inequalities. We use `FindInstance` for refutations after collecting suitable equations and inequalities in the antecedent and consequent, see section 5.2 below for an application.
- `GroebnerBasis` is an implementation of Buchberger's Gröbner basis algorithm. We use the algorithm to derive contradictions among sets of polynomial equations. This method is employed in the proof of associativity of the group law for rational points on an elliptic curve.
- `FullSimplify` is a general purpose simplifier which tries a wide range of transformations. It is employed frequently to simplify algebraic expressions in the consequent, using information from the antecedent as assumptions.

Since the exact theory, in which some of those *Mathematica*-algorithms are valid, is in some cases not spelled out, it is impossible to formally specify the theory w.r.t. which certain Analytica-proofs are valid. Some computer-algebra-based proof steps, however, are provenly correct, and in these cases the underlying theory is well-known. As an example, take the reduction of proving the unsolvability of a system of polynomial equations to checking its reduced Gröbner basis to be $\{1\}$, which is proven to be true in the theory of multivariate polynomials over some field. Of course, we rely/depend on the correct implementation of the Gröbner basis algorithm in *Mathematica*, but this does not differ from using one's own implementation. In other cases the situation is unfortunately much less clear, e.g. when using `FullSimplify`, we need to trust that what *Mathematica* returns as a simplified expression is equal/equivalent to the original expression (w.r.t. some theory) as claimed by the *Mathematica*-manual. At least, as mentioned in Section 3.2.2, when

converting expressions from Analytica to *Mathematica* we explicitly check certain side-conditions, which are assumed by *Mathematica* implicitly.

4.5 The Handling of Definitions

We support both *explicit definitions* of predicates and functions and *implicit function definitions*. As a special language construct, a *case distinction* may be used define functions or predicates by cases. Accordingly, if functions or predicates defined by cases are present in a proof, we provide proof rules to split the proof into the respective cases. Moreover, we employ predicate definitions when building up the look-up table, see Section 3.2.1.

5 Examples

5.1 Group Characters

The proof of Dirichlet's Theorem refers to the concept of number-theoretic group characters. For an abelian multiplicative group G a *group character* is a homomorphism from $(G, \cdot)$ to $(\mathbb{C}^*, \cdot)$, where $\mathbb{C}^*$ denotes the multiplicative subgroup of the complex numbers. The set $GC[G]$ of group characters on G has a natural multiplication, namely, $(a \cdot b)(x) = a(x)b(x)$.

Theorem 5.1 *$GC[G]$ is an abelian multiplicative group with the identity character $\mathbf{1}(x) = 1$ and the inverse character*[10] *reciprocal$(a)(x) = a(x)^{-1}$.*

The prover configuration for a completely automated proof of this theorem is as follows: The rulebase contains a rule for expanding definitions in the consequent, a rule for verifying equations in the consequent, the logical rules for And-Right and Or-Left as described in Section 2.3, and the standard normalization rule. For building up the look-up table as described in Section 3.2.1, we use all algebraic definitions known to Analytica, which define semigroups, monoids, groups,..., homomorphisms, group characters hierarchically. We will show parts related to proving the inverse-property in more detail. For the annotated presentation of the automated proof below we use abbreviations for function- and predicate symbols introduced in the formalization of the theorem: AMG$[G, \mathbf{1}, \mathbf{i}]$ formalizes "G is an abelian multiplicative group with identity $\mathbf{1}$ and inverse function $\mathbf{i}$".

Proof.

$\forall_{\text{inv}}\forall_{\text{id}}\forall_G$ AMG$[G, \text{id}, \text{inv}] \Rightarrow$
AMG$[\text{GC}[G], \mathbf{1}, \text{reciprocal}]$

Theorem: If G is an abelian multiplicative group then also the group characters on G form an abelian multiplicative group.

[10] Note the currying in the definition of both multiplication and inverse of group characters, see Section 2.1.

We have to show:

$$\mathrm{AMG}[\mathrm{GC}[G_2], \mathbf{1}, \mathrm{reciprocal}]$$

$$\vdots$$

We have to show:

$$\mathrm{isInverse}[\mathrm{reciprocal}, \mathrm{times}, \mathbf{1}, \mathrm{GC}[G_2]]$$

We expand definitions in the goal, thus, the new proof goal is

$$\forall_{a \in \mathrm{GC}[G_2]} a^{-1} \in \mathrm{GC}[G_2] \wedge$$

$$\forall_{a \in \mathrm{GC}[G_2]} a *_{\mathrm{GC}[G_2]} a^{-1} =_{\mathrm{GC}[G_2]} \mathbf{1}$$

$$\vdots$$

$$(a_1^{-1})[x_1] *_{\mathbb{C}^*} (a_1^{-1})[y_1] =_{\mathbb{C}^*}$$

$$(a_1^{-1})[x_1 *_{G_2} y_1].$$

Function definitions expand inside the goal, so we get

$$(a_1[x_1])^{-1} *_{\mathbb{C}^*} (a_1[y_1])^{-1} =_{\mathbb{C}^*}$$

$$(a_1[x_1 *_{G_2} y_1])^{-1}.$$

Custom rewrite rules apply in order to simplify the sequent.
We are left with

$$(a_1[x_1])^{-1} *_{\mathbb{C}^*} (a_1[y_1])^{-1} =_{\mathbb{C}^*}$$

$$(a_1[x_1] *_{\mathbb{C}^*} a_1[y_1])^{-1}.$$

During conversion from an Analytica expression to its corresponding *Mathematica* expression we need to check

$$a_1[x_1] *_{\mathbb{C}^*} a_1[y_1] \neq 0$$

but this follows easily from

Normalization skolemizes the universal quantifiers and splits the implication.
Stepwise expanding definitions. Split conjunction in consequent. Closure, associativity, identity-property all proved. Finally, we expand the definition of being an inverse. Note, that all operations carry information about their domain, we denote it as a subscript. Now, the conjunction will be split, and we concentrate on the first part, i.e. that the inverse character in fact *is a group character* on G_2. Again, skolemization and expansion of $a_1 \in \mathrm{GC}[G_2]$, which means, a_1 must be a homomorphism from G_2 into $\mathbb{C}^*$.

Several properties have to be proven: a_1^{-1} must first of all be a mapping, etc. we skip forward to the interesting property that the inverse character a_1^{-1} has the homomorphism property. Note that, in a previous simplification attempt, the left and right side of the equality have been swapped. Now the *definition of the inverse character* applies on both sides, essentially pulling the $^{-1}$ outside. On the right side we can now use the property that a_1 is a homomorphism. Of course, the simplification rule for homomorphisms is implemented in such a way that it *verifies* that a_1 actually *is a homomorphism*, which is easy by look-up since we know a_1 to be a group character on G_2. We are left with an equality on $\mathbb{C}$, which we simply ship to *Mathematica*. During conversion from Analytica to *Mathematica* we check side-conditions, notably the expressions to be inverted must be unequal zero! In this case, we look up an expression of the form $x * y \neq 0$, and the look-up proceeds by looking up $x * y \in \mathrm{forsome}[D]^*$. Now, $x * y \in D$ will look up $\mathrm{isClosed}[*, D] \wedge x \in D \wedge y \in D$,

$$\begin{array}{c}\text{AMG}[\mathbb{C}^*, 1, \text{reciprocal}],\\ a_1 \in \text{GC}[G_2],\\ x_1 \in G_2,\\ y_1 \in G_2.\end{array}$$

The equality can be proved by simplification.

$$\vdots$$

where the search will branch. Looking up isClosed$[*, D]$ will go through the entire algebraic hierarchy through semigroup, monoid, etc. finally succeeding through built-in knowledge from the prover configuaration that $\mathbb{C}^*$ is an abelian multiplicative group. $a_1[x_1], a_1[y_1] \in \mathbb{C}^*$ it will find since a_1 is a group character on G_2 and $x_1, y_1 \in G_2$.

The remaining parts of the proof use the same techniques as already illustrated. □

The issue worth mentioning is that we need not expand definitions in the antecedent but still have certain hidden knowledge inferable from the antecedent available in the proof. For example, we certainly do not wish to expand the fact that $\mathbb{C}^*$ is an abelian multiplicative group down to the property that $\mathbb{C}^*$ is closed under multiplication. This becomes even more important when we think about dealing with large knowledge bases. Analytica can also prove a more general form of this theorem: the set of homomorphism (with multiplication like group characters) between two abelian groups forms a group. The proof goes along the same lines, except that we cannot send equalities to *Mathematica* to be verified. Here, the group simplifiers described in Section 4 come into play, and they have no problems with the group simplifications needed in these examples.

5.2 The Weak Mordell-Weil Theorem

The Weak Mordell-Weil Theorem states that, for any elliptic curve $E(\mathbb{Q})$, the quotient group $E(\mathbb{Q})/2E(\mathbb{Q})$ is finite. We consider the proof of this fact for elliptic curves of the form $y^2 = (x-a)(x-b)(x-c)$ where a, b, c are distinct rational numbers. The proof, as given in [KKS00], proceeds as follows.

We define a mapping $\delta : E(\mathbb{Q}) \to \mathbb{Q}^*/(\mathbb{Q}^*)^2 \times \mathbb{Q}^*/(\mathbb{Q}^*)^2 \times \mathbb{Q}^*/(\mathbb{Q}^*)^2$ by:

$$\delta(P) = \begin{cases} (1,1,1) & \text{if } P = O,\\ ((a-b)(a-c), a-b, a-c) & \text{if } P = (a,0),\\ (b-a, (b-a)(b-c), b-c) & \text{if } P = (b,0),\\ (c-a, c-b, (c-a)(c-b)) & \text{if } P = (c,0),\\ (x-a, x-b, x-c) & \text{otherwise, } P = (x,y). \end{cases}$$

The proof of the theorem comprises three parts:

(i) δ is a homomorphism.

(ii) The kernel of δ is $2E(\mathbb{Q})$.

(iii) The image of delta is contained in the finite subgroup generated by the

prime factors of $a-b$, $b-c$, $c-a$, -1.

By the first isomorphism theorem, we then have that $E(\mathbb{Q})/2E(\mathbb{Q})$ is isomorphic to a finite group. Our status in automating each of these pieces is as follows:

(i) This part has been automated in the most general case, i.e. for points in general positions. Missing for full automation of the other cases are some simplification mechanisms.

(ii) For this part, we consider a decomposition of the multiplication-by-two map into functions g, h, such that $h \circ g$ is the map. From this decomposition it is easily seen that the image of the map is contained in the kernel of δ. The reverse containment is implied by the surjectivity of g from $E(Q)$ onto a special domain, and the surjectivity of h from that domain onto the kernel of δ, both of which we have automated.

(iii) We have fully automated this part. Analytica handles the case-splitting correctly and takes care of each case.

Here we present a shortened and edited proof of the third part, as generated by Analytica. Except for pretty-printing we have retained most of the Analytica syntax. Thus, image$[f, D]$ denotes the image of f over domain D. $\delta[a, b, c]$ refers to the map δ associated with the curve parameters a, b and c. elemGen $[s, *, G, P]$ means that element s is generated from P in group $\langle G, * \rangle$. We write pV$[p, q]$ for the p-adic valuation of q.

Theorem 5.2

$$\forall_{a \in \mathbb{Q}} \left(\forall_{b \in \mathbb{Q}} \left(\forall_{c \in \mathbb{Q}} image \left[\delta[a, b, c], E(Q) \right] \subseteq G \times G \times G \right) \right)$$

where

$$G = \{\, s \mid elemGen \left[s, times, \left(\mathbb{Q}/\mathbb{Q}^2 \right) \times \left(\mathbb{Q}/\mathbb{Q}^2 \right) \times \left(\mathbb{Q}/\mathbb{Q}^2 \right), P \right] \}$$

and

$$\begin{aligned} P = \{ p\mathbb{Q}^2 \mid\ & p =_{\mathbb{Z}} -1 \vee pV[p, a -_{\mathbb{Q}} b] \neq_{\mathbb{Z}} 0 \vee \\ & pV[p, b -_{\mathbb{Q}} c] \neq_{\mathbb{Z}} 0 \vee pV[p, c -_{\mathbb{Q}} a] \neq_{\mathbb{Z}} 0 \} \end{aligned}$$

Proof. After expanding the definitions the first interesting event is the following new formula in the antecedent.

$$\mathrm{s} =_{(\mathbb{Q}/\mathbb{Q}^2)^3} \text{vector}\,[1, 1, 1] \vee \mathrm{s} \in \{ \delta(x, y) | x, y \in \mathbb{Q}, y^2 =_{\mathbb{Q}} (x-a)(x-b)(x-c) \}$$

The case where $s = (1, 1, 1)$ is easily taken care of. Next we consider the image of δ. The case distinction mechanism first tackles $x =_{\mathbb{Q}} a$. The crucial event associated with this case is the following formula in the consequent:

$$\text{elemGen}\left[(a -_{\mathbb{Q}} b) *_{\mathbb{Q}} (c -_{\mathbb{Q}} a), *, \mathbb{Q}/\mathbb{Q}^2, P\right] \wedge \text{elemGen}\left[a -_{\mathbb{Q}} b, *, \mathbb{Q}/\mathbb{Q}^2, P\right] \\ \wedge \text{elemGen}\left[c -_{\mathbb{Q}} a, *, \mathbb{Q}/\mathbb{Q}^2, P\right]$$

This case is easily handled since $(a-b)(a-c)$, $a-b$, $a-c$ are trivially generated by P. The cases $x =_{\mathbb{Q}} b$ and $x =_{\mathbb{Q}} c$ are similar. The remaining generic case has additional assumptions $x \neq_{\mathbb{Q}} a, x \neq_{\mathbb{Q}} b, x \neq_{\mathbb{Q}} c$. The key event here is

$$\text{elemGen}\left[x -_{\mathbb{Q}} a, *, \mathbb{Q}/\mathbb{Q}^2, P\right] \wedge \text{elemGen}\left[x -_{\mathbb{Q}} b, *, \mathbb{Q}/\mathbb{Q}^2, P\right] \\ \wedge \text{elemGen}\left[x -_{\mathbb{Q}} c, *, \mathbb{Q}/\mathbb{Q}^2, P\right]$$

Exploiting symmetry, the prover now establishes only the assertion

$$\text{elemGen}\left[x -_{\mathbb{Q}} a, *, \mathbb{Q}/\mathbb{Q}^2, P\right]$$

Using an auxiliary lemma [11] and backchaining the proof goal is now translated into an assertion about p-adic valuations: from assumptions

$$p \in \text{Primes}, \text{pV}\left[p, x -_{\mathbb{Q}} a\right] \notin 2\mathbb{Z}.$$

we need to establish the goals

$$\text{pV}\left[R, a -_{\mathbb{Q}} b\right] \neq_{\mathbb{Z}} 0, \text{pV}\left[R, b -_{\mathbb{Q}} c\right] \neq_{\mathbb{Z}} 0, \text{pV}\left[R, c -_{\mathbb{Q}} a\right] \neq_{\mathbb{Z}} 0.$$

To this end we use properties of p-adic valuations stored in the knowledge base:

$$\text{pV}\left[p, s -_{\mathbb{Q}} t\right] \geq_{\mathbb{Q}} \min[\mathbb{Z}, \text{pV}[p, s], \text{pV}[p, t]] \\ \text{pV}[p, s] \neq_{\mathbb{Q}} \text{pV}[p, t] \Rightarrow \text{pV}\left[p, s -_{\mathbb{Q}} t\right] =_{\mathbb{Q}} \min[\mathbb{Z}, \text{pV}[p, s], \text{pV}[p, t]]$$

By heuristic instantiation we now generate additional assumptions such as

$$\min[\mathbb{Q}, \text{pV}[p, -a + x], \text{pV}[p, -b + x]] \leq \text{pV}[p, a - b].$$

and

$$\text{pV}[p, x - a] =_{\mathbb{Q}} \text{pV}[p, c - a] \vee \text{pV}[p, x - c] =_{\mathbb{Q}} \min[\mathbb{Q}, \text{pV}[p, c - a], \text{pV}[p, x - a]].$$

At this point the prover uses `FindInstance`, a *Mathematica* decision algorithm for the existence of solutions of systems of equations and inequalities, to refute these additional premises and the negated goals.

□

[11] The supplied auxiliary lemma states that if for all primes p such that $\text{pV}[p, t]$ is odd, we have $p \in S$, and $-1 \in S$, then S generates $t\mathbb{Q}^2$ in $\mathbb{Q}^*/(\mathbb{Q}^*)^2$

6 Conclusion

The work reported in this paper is work in progress towards a fully automated proof of the Mordell-Weil Theorem and a theorem of Dirichlet. The proof search, the look-up mechanism, and the handling of definitions are completely general, however, and their interplay behaved very promising in the examples done up to now. In particular, the fully automated setup of a proof by case distinction based on definitions by cases will be applied to proving the associativity of the group law on elliptic curves, a symbolic computation proof, which has probably newer been done up to now due to its many cases to distinguish.

References

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

[CKOS03] E. Clarke, M. Kohlhase, J. Ouaknine, and K. Sutner. Analytica 2. In *Calculemus, Rome*, 2003.

[CZ92] Edmund Clarke and Xudong Zhao. Combining symbolic computation and theorem proving: Some problems of ramanujan. In D. Kapur, editor, *Proceedings the 11th Conference on Automated Deduction*, volume 607 of *LNCS*, pages 66–78, Saratoga Spings, NY, USA, 1992. Springer Verlag.

[KKS00] Kazuya Kato, Nobushige Kurokawa, and Takeshi Saito. *Number Theory 1: Fermat's Dream*. Translations of Mathematical Monographs. American Mathematical Society, 2000. ISBN 0-821-80863-X.

[Sil86] Joseph H. Silverman. *The Arithmetic of Elliptic Curves*. Graduate Texts in Mathematics. Springer-Verlag, 1986. ISBN 0-387-96203-4.

[Win99] W. Windsteiger. Building Up Hierarchical Mathematical Domains Using Functors in Theorema. In A. Armando and T. Jebelean, editors, *Electronic Notes in Theoretical Computer Science*, volume 23-3, pages 83–102. Elsevier, 1999. Calculemus 99 Workshop, Trento, Italy.

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

Theorema 2.0: A Graphical User Interface for a Mathematical Assistant System

Wolfgang Windsteiger

RISC, JKU Linz
4232 Hagenberg, Austria

Wolfgang.Windsteiger@risc.jku.at www.risc.jku.at/home/wwindste/

Theorema 2.0 stands for a re-design including a complete re-implementation of the *Theorema* system, which was originally designed, developed, and implemented by Bruno Buchberger and his *Theorema* group at RISC. In this paper, we present the first prototype of a graphical user interface (GUI) for the new system. It heavily relies on powerful interactive capabilities introduced in recent releases of the underlying Mathematica system, most importantly the possibility of having dynamic objects connected to interface elements like sliders, menus, check-boxes, radio-buttons and the like. All these features are fully integrated into the Mathematica programming environment and allow the implementation of a modern user interface.

1 Introduction

The reasons for re-designing the *Theorema* system are manifold. Some of them refer to the usability of the system from the users' point of view, others are related to the flexibility from the developers' point of view such as limitations in the combination of provers. Finally, using the system and observing other users working with the system over the time has shown several possibilities for improvements that cannot be easily realized without reconsideration of principal design decisions which have been made fifteen years ago. For these reasons, we decided to re-implement *Theorema* with the aim to re-design those components, that have turned out to be the main hurdles, and reuse design ideas[1] that have proved successful and useful.

A crucial decision in every software development project is, of course, the choice of the development platform. *Theorema 1.0* was based on Mathematica [8], one of the leading computer algebra systems developed by Wolfram Research, mainly for two reasons: firstly, because of its convenient programming language offering the powerful pattern matching mechanism, which is extremely well-suited for the implementation of logical inference rules, and secondly for the nice notebook user interface. The availability of a huge library of symbolic mathematical algorithms does not harm, but it is not and it never was the crucial factor in favor of Mathematica. Maybe the only drawback is the commercial setting, a *Theorema* user needs to purchase a license of Mathematica in order to be able to run *Theorema*. But there are additional arguments on the pro-side, such as platform independence, i.e Mathematica programs run without any modifications on essentially all available operating system platforms (Linux, OS X, and Windows), the powerful development group at Wolfram Research that keeps Mathematica being always an up-to-date platform growing into various directions, and the huge group of Mathematica users

[1] What we reuse are *the ideas* only. We do not reuse any existing Mathematica code from *Theorema 1.0*, because we want to change some internal datastructures and cleanup the code base on that occasion as well.

C. Kaliszyk and C. Lüth (Eds.): 10th International Workshop
on User Interfaces for Theorem Provers
EPTCS 118, 2013, pp. 72–82, doi:10.4204/EPTCS.118.5

also outside the classical theorem proving community. We did have minor compatibility issues when new releases of Mathematica arrived, but in retrospect we probably had fewer problems over more than fifteen years than we would have had on comparable platforms. We therefore decided to stay with Mathematica also for the implementation of *Theorema 2.0*. In this paper, we concentrate on the novelties related to the user interface exclusively.

Theorema 1.0 [1, 2, 3, 7] has been widely acknowledged as a system with one of the nicer user interfaces. However, we could observe that outsiders or beginners still had a very hard time to successfully use the *Theorema* system. This was true for entering formulae correctly as well as for proving theorems or performing computations. The principal user interface to *Theorema* is given by the Mathematica notebook front-end, giving a small advantage to Mathematica users as they are familiar with the main interaction patterns offered by the notebook interface. While the 2D-syntax for mathematical formulae available since Mathematica 3 is nice to read, a wrongly entered 2D-structure has always been a common source of errors. More than that, the user-interaction paradigm in *Theorema 1.0* was the standard 'command-evaluate' known from Mathematica, meaning that every action in *Theorema 1.0* was triggered by the evaluation of a certain *Theorema* command implemented as a Mathematica program. As an example, giving a definition meant evaluation of a Definition[...]-command, stating a theorem meant evaluation of a Theorem[...]-command, proving a theorem meant evaluation of a Prove[...]-command, and performing a computation meant evaluation of a Compute[...]-command. For the new *Theorema 2.0* system, we envisage a more 'point-and-click'-like interface as one is used to from modern software tools like a mail user agent or office software.

The main target user group for *Theorema* are mathematicians, who want to engage in formalization of mathematics or who just want to have some computer-support in their proofs. The system should be a tool helping to grasp the nature of proving, thus, students of mathematics or computer science are typical users as well as teachers at universities or high schools. For the latter groups in particular, nice two-dimensional input and output of formulae in an appearance like typeset or handwritten mathematics is an important feature. On the other hand, the unambiguous parsing of mathematical notation is non-trivial already in 1D, supporting 2D-notations introduces some additional difficulties.

Theorema is a multi-method system, i.e. it offers many different proving methods specialized for the proof task to be carried out. The main focus lies on a resulting proof that comes as close as possible to a proof done by a well-educated mathematician. This results in a multitude of methods, each of them having a multitude of options to fine-tune the behavior of the provers. This is on the one hand powerful and gives many possibilities for system insiders, who know all the tricks and all the options including the effect they will have in a particular example. For newcomers, on the other hand, the right choice of an appropriate method and a clever choice of option settings is often an insurmountable hurdle. The user interface in *Theorema 2.0* makes these selections easier for the user. Furthermore, the user has the possibility to extend the system by self-defined reasoning rules and strategies.

Finally, the integration of proving, computing, and solving *in one system* will stay a major focus also in *Theorema 2.0*. Compared to *Theorema 1.0*, the separation between *Theorema* and the underlying Mathematica system is even stricter, but the integration of Mathematica's computational facilities into the *Theorema* language has been improved.

Theorema 2.0 is currently under development. The components described in this paper are all implemented and the screenshots provided show a running and working system, it is not the sketch of a design. However, the interface presented here is incomplete and it will grow with new demands. From the experience with Mathematica's GUI components gathered up to now we are confident that all requirements for a modern interface to a mathematical assistant system can easily be fulfilled based on that platform. Some of the features described in this paper rely or depend on their implementation in Mathematica.

This requires a certain knowledge of the principles of Mathematica's programming language and user front-end in order to understand all details given below. The rest of the paper is structured as follows: the first section describes the new features in recent releases of Mathematica that form the basis for new developments in *Theorema 2.0*, in the second section we introduce the new *Theorema* user interface, and in the conclusion we give a perspective for future developments.

2 New in Recent Versions of Mathematica

In this section, we describe some of the new developments in recent Mathematica releases that were crucial in the development of *Theorema 2.0*.

2.1 Mathematica Dynamic Objects

Earlier versions of Mathematica offered the so-called *GUIKit* extension, which was based on Java and used MathLink for communication between Mathematica and the generated GUI. We used GUIKit earlier for the development of an educational front-end for *Theorema* [6], but the resulting GUI was cumbersome to program, unstable, and slow in responding to user interaction. As of Mathematica version 6, and then reliably in version 7, the concept of *dynamic expressions* was introduced into the Mathematica programming language and fully integrated into the notebook front-end. Dynamic expressions form the basis for interactive system components, thus, they are *the* elementary ingredient for the new *Theorema 2.0* GUI.

In short, every Mathematica expression can be turned into a dynamic object by wrapping it into `Dynamic`. As the most basic example, `Dynamic[`*expr*`]` produces an object in the Mathematica front-end that displays as *expr* and automatically updates as soon as the value *expr* changes. In addition, typical interface elements such as sliders, menus, check-boxes, radio-buttons, and the like are available. Every such element can then be connected to a program variable, such that user interactions (e.g. clicking a check-box) are reflected in the values of the respective variables. The set of available GUI objects is very rich and there is a wide variety of options and auxiliary functions in order to influence their behavior and interactions. These features allow the construction of arbitrarily complicated dynamic interfaces and seem to constitute a perfect platform for the implementation of an interface to the *Theorema* system. A big advantage of this approach is that the entire interface programming can be done inside the Mathematica environment, which in particular brings us a uniform interface on all platforms from Linux over Mac until Windows for free.

2.2 Cascading Stylesheets

Stylesheets are a means for defining the appearance of Mathematica notebook documents very similar to how stylesheets work in HTML or word processing programs. The mere existence of a stylesheet mechanism for Mathematica notebooks is not new, but what is new since version 6 is that stylesheets are cascading, i.e. stylesheets may depend on each other and may inherit properties from their underlying styles just like CSS in HTML. This of course facilitates the design of different styles for different purposes without useless duplication of code. The more important news is that stylesheets can now, in addition to influencing the appearance of a cell in a notebook, also influence the *behavior* of a cell. This is a feature that we always desired since the beginning of *Theorema*: an action in Mathematica is always connected in some way to the evaluation of a cell in a notebook, and we wanted to have different evaluation behavior depending on whether we want to e.g. prove something, do a computation, enter a formula,

or execute an algorithm. Using a stylesheet, we can now define computation-cells or formula-cells, and the stylesheet defines commands for their pre-processing, evaluation, and post-processing.

3 The *Theorema* Interface

The Mathematica notebook front-end is the primary user interface for *Theorema*. "Working in *Theorema*" consists of *activities* that themselves require certain *actions* to be performed. As an example, a typical activity would be "to prove a theorem", which requires actions such as "selecting a proof goal", "composing the knowledge base", "choosing the inference rules and a proof strategy", etc. The central new component in *Theorema 2.0* is the *Theorema commander*; it is the GUI component that guides and supports all activities and actions. Of course, most activities work on mathematical formulae in one or the other way. Formulae appear as definitions, theorems or similar *environments* and are just written into Mathematica/*Theorema* notebook documents that use one of the *Theorema* stylesheets. As *Theorema session* we refer to the collection of all formulae passed to the system up to a certain moment. Composing and manipulating the session is just another activity and therefore supported from the *Theorema* commander. The second new interface component in *Theorema 2.0* is the *virtual keyboard*; its task is to facilitate the input of math expressions, in particular 2D-input. Figure 1 shows a screen shot of *Theorema 2.0* with a *Theorema*-styled notebook[2] top-left, the *Theorema* commander to its right, and the virtual keyboard underneath. Of course, all these features are just add-ons to the standard Mathematica interface, thus, support for notebook formatting, inputting special characters, text styling, and the like through notebook menus, palettes, and/or keyboard shortcuts need not be implemented from the scratch.

3.1 Organizing a *Theorema* Session

When working in *Theorema* one composes one or more *Theorema*-styled Mathematica notebooks, which have all the capabilities of normal Mathematica notebooks plus the possibility to process expressions in *Theorema* language inside so-called *formula cells*. This means that *Theorema* expressions are embedded in a full-fledged document format for mathematical writing. Mathematica notebooks consist of hierarchically arranged cells, whose nesting is visualized with cell brackets on the right margin of the notebook. Figure 1 shows a notebook using one of the *Theorema*-specific stylesheets responsible for the notebook's appearance and behavior. Note in particular that, due to this stylesheet, each environment forms a group for its own.

Theorema formula cells contain mathematical expressions in *Theorema* syntax with an additional label. If no label is given by the user, a numerical label, which is unique within the notebook, is automatically assigned. User-supplied labels need not be unique, but the system issues a warning to the user. As soon as the formula is passed to the system through Mathematica's standard Shift-Enter, the formula is stored in an internal datastructure that carries a *unique key* for each formula in addition to the formula itself and its label. This key consists of the absolute pathname of the notebook file in which it was given, and the unique cell-ID within that notebook, which is provided by the Mathematica front-end. In formula display, we typically use the label, but when actually referring to a formula in the interface, we use the unique formula key. As we will explain later, the user never sees nor needs the concrete formula key explicitly.

[2] The actual mathematics written in the notebook is irrelevant for this paper, but for the curious it is part of a formalization of auction theory, an important application of mathematics in economy. This is joint work with M. Kerber, C. Lange, and C. Rowat at the University of Birmingham [5].

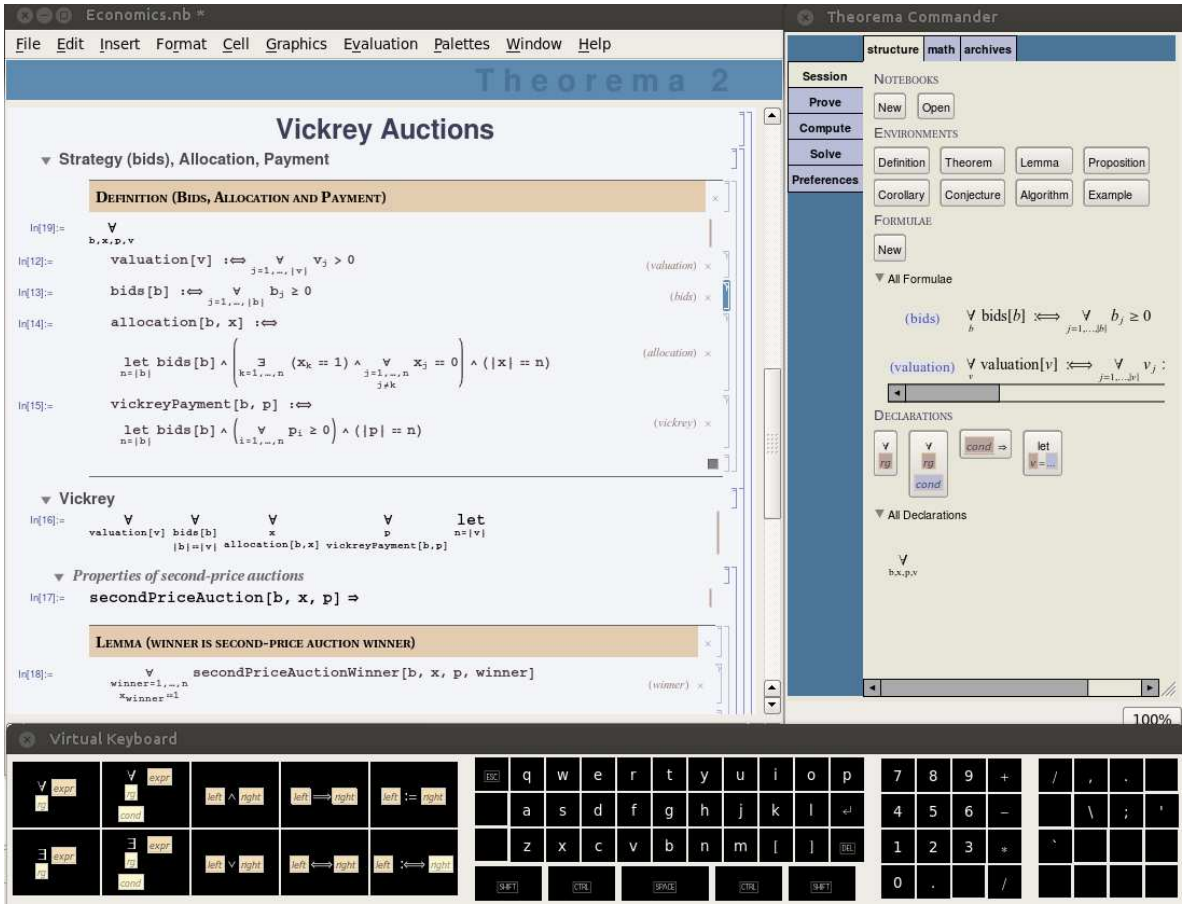


Figure 1: The *Theorema 2.0* GUI

In mathematical practice, universal quantification of formulae and conditioning is often done on a global level. As an example take definitions, which often start with a phrase like "Let $n \in \mathbb{N}$. We then define ...", which in effect expresses a universal quantifier for n plus the condition $n \in \mathbb{N}$ for all notions introduced in the current definition. For this purpose, we provide *global declarations*, which may either contain one or several "orphaned" universal quantifiers (each containing a variable and an optional condition, but missing the formula, to which they refer) or an "orphaned" implication (missing its right hand side), or an abbreviation indicated by a "let". The idea is that the scope of such a declaration ranges to the end of the environment in which it appears. In the example in Figure 1, this is used in Definition (Bids, Allocation and Payment) with a universal quantifier for b, x, p, and v valid for all formulae inside that definition. When passing to the system e.g. the formula written in the notebook as

$$\text{bids}[b] :\Longleftrightarrow \forall_{j=1,\ldots,|b|} b_j \geq 0$$

it actually results in

$$\forall_b \text{bids}[b] :\Longleftrightarrow \forall_{j=1,\ldots,|b|} b_j \geq 0$$

being stored in the *Theorema* session. For the user's convenience, the *Theorema* commander always shows all formulae currently available in the section labeled 'All Formulae' as shown in Figure 1. There

one can also see, that quantifiers are of course only put for those variables that actually appear free in the formula. The cell grouping defined in the stylesheet ensures that a definition gets its own cell group that limits the scope of the quantifier.

We generalized this idea so that a global declaration can be put anywhere in a notebook, and its scope ranges similar to the situation described above from its position to the end of the nearest enclosing cell group. In Figure 1, this is used twice:

1. There is a big
$$\underset{\text{valuation}[v]}{\forall}\ \underset{|b|=|v|}{\underset{\text{bids}[b]}{\forall}}\ \underset{\text{allocation}[b,x]}{\forall x}\ \underset{\text{vickreyPayment}[b,p]}{\forall p}\ \underset{n=|v|}{\text{let}}$$
at the beginning of Section 'Vickrey'. This means that, without further mentioning, all free occurrences of $v, b, x,$ and p will be universally quantified with the respective additional conditions in the entire section including all its subsections. Furthermore, wherever we write n it is just an abbreviation for $|v|$.
2. There is a 'secondPriceAuction$[b,x,p] \Rightarrow$' in Subsection 'Properties of second-price auctions', so that this condition on $b, x,$ and p affects only the rest of this subsection.

At the moment of passing a formula to the system, all declarations valid at this position are silently applied and the actual formula in the *Theorema* session has all intended quantifiers and conditions attached to it just as if they were written explicitly with the formula. Thus, the Lemma compactly written as

$$\underset{\substack{\text{winner}=1,\ldots,n\\ x_{\text{winner}}=1}}{\forall} \text{secondPriceAuctionWinner}[b,x,p,\text{winner}]$$

in the notebook in Figure 1 actually states

$$\underset{\text{valuation}[v]}{\forall}\ \underset{|b|=|v|}{\underset{\text{bids}[b]}{\forall}}\ \underset{\text{allocation}[b,x]}{\forall x}\ \underset{\text{vickreyPayment}[b,p]}{\forall p}$$
$$\text{secondPriceAuction}[b,x,p] \Rightarrow \underset{\substack{\text{winner}=1,\ldots,|v|\\ x_{\text{winner}}=1}}{\forall} \text{secondPriceAuctionWinner}[b,x,p,\text{winner}].$$

This is quite convenient and comes very close to how mathematicians are used to write down things. In essence, the effect of global declarations is similar to what can be achieved with contexts or locales in Isabelle [4]. For bigger documents, however, one might lose the overview on which declarations are valid at a certain point in a notebook. The *Theorema* commander gives valuable assistance in this situation: the section labeled 'All Declarations' always shows all declarations valid at the current cursor position in the selected notebook. In Figure 1, the selection is at the cell containing the definition of bids$[b]$ within the Definition-environment, and correspondingly the *Theorema* commander displays the $\underset{b,x,p,v}{\forall}$ valid there.

3.2 The *Theorema* Commander

Figure 1 top-right shows the *Theorema* commander, the main GUI component in *Theorema 2.0*. It is a two-level tabview structured according to activities on the first level and the corresponding actions for each activity on the second level. The first-level activity-tabs reside on the left margin of the *Theorema* commander. Currently, the supported activities are 'Session', 'Prove', 'Compute', 'Solve', and 'Preferences', but as the system develops, this list may increase. For each of these activities, the respective

actions can be accessed on the top margin of the *Theorema* commander. Moving through them from left to right corresponds to a wizard guiding the user through the respective activity. Proving is presumably the most interesting activity and we will therefore elaborate it in more detail in the next paragraph. The remaining parts of the *Theorema* commander are of similar fashion, we will only mention some highlights in the concluding paragraph of this section.

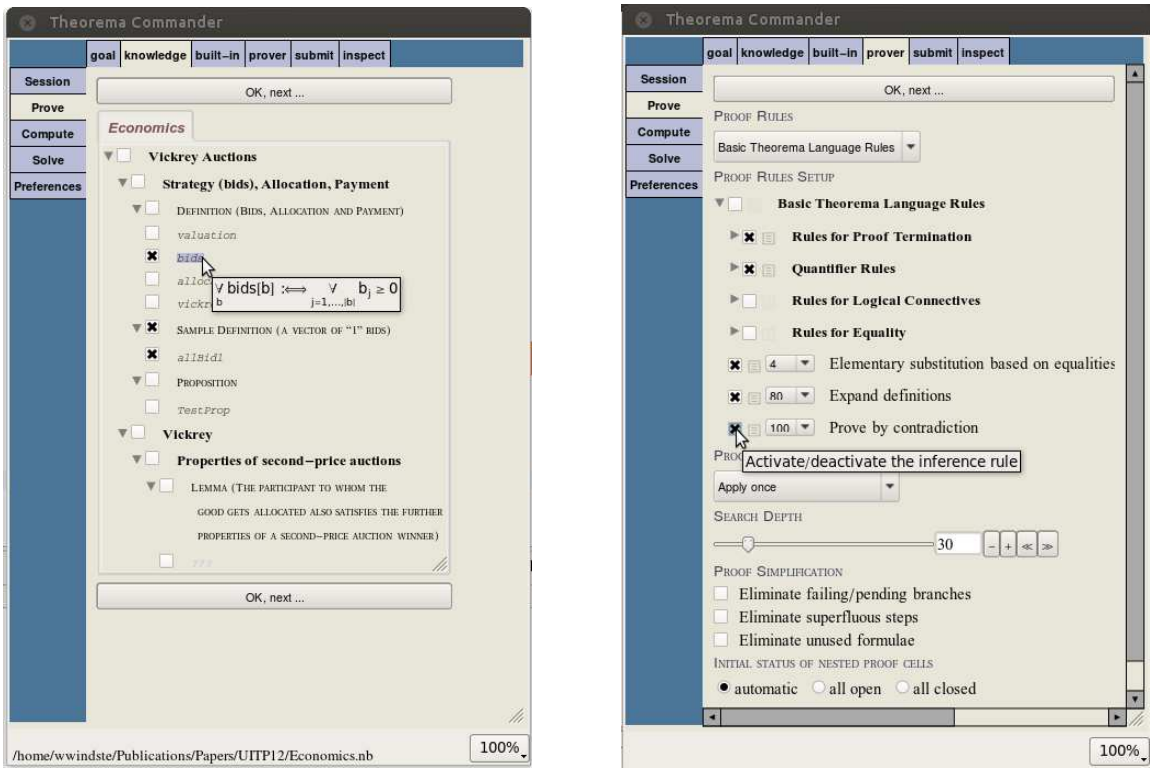


Figure 2: The 'Prove'-activity: the knowledge browser (left) and the prover configuration (right).

The 'Prove'-activity The 'Prove'-activity consists of actions 'goal', 'knowledge', 'built-in', 'prover', 'submit', and 'inspect', see the screenshots in Figure 2. These action correspond to the individual steps when proving a theorem in *Theorema*: it requires the specification of the proof goal, the specification of the knowledge available in the proof, setting up built-in knowledge, and selecting/configuring the desired prover to be used. After submitting the proof problem to *Theorema*, the system will show a successful or failing proof, which the user can then inspect.

Defining the proof goal is as simple as just selecting a cell containing the formula to be proved in an open notebook with the mouse. The selected formula is then shown in the 'goal'-tab, and it changes with every mouse selection. The only action required here is to confirm the choice by pressing a button in the 'goal'-tab. From this moment on, the proof goal is fixed until the next confirmation, whatever the mouse selects.

Then the user needs to compose the knowledge base to be used in the proof, see Figure 2 (left). The *knowledge browser* displays a tab for each open notebook or loaded knowledge archive[3]. In each tab, a

[3] *Archives* are another new development in *Theorema 2.0*. An archive gives the possibility to store the formulae from a

hierarchical overview of the file/archive content is displayed, showing only the section structure, environments, and formula labels. Simply moving the mouse cursor over the label opens a tooltip displaying the whole formula, clicking the label jumps to the respective position in the corresponding notebook/archive. Each entry in the browser has a check-box attached to its left responsible for toggling the selection of the respective unit. In this way, individual formulae, environments, sections, up to entire notebooks can be selected or deselected with just one mouse-click. The formulae chosen in this way constitute the knowledge base for the next proof. The formula label displayed in the browser is only syntactic sugar, the check-box is connected to the unique key of each formula in the *Theorema* session, see Section 3.1.

The next action within the 'Prove'-activity is the selection of built-in computational knowledge[4]. The *built-in browser* works like the knowledge browser described above. Instead of section grouping we have (not necessarily disjoint) thematic groups of built-ins like sets, arithmetic, or logic. Built-in knowledge is applied in proving in order to simplify formulae by computation on finite objects, e.g. computations with numbers or finite sets. We do not go into further details.

After having composed the relevant built-in knowledge, the user needs to select the prover. A *prover* in *Theorema 2.0* consists of a (possibly nested) list of inference rules accompanied with a proof strategy. Accordingly, the 'prover'-action shows menus for choosing the inference rules and the strategy, respectively, together with short info panels explaining the current choice as depicted in Figure 2 (right). The 'prove'-action displays an *inference rule browser* corresponding to the selected rule list. Its functionality is like that of the knowledge browser described above, only that it is using the nesting structure of the inference rule list for setting up the hierarchy, which gives the possibility to activate/deactivate entire groups with only one click. Using the inference rule browser the user can efficiently deactivate individual (groups of) inference rules, e.g. for influencing whether an implication will be proved directly or via contraposition. In addition to the checkbox for activation and deactivation, the interface allows to decide whether the respective proof step should be explained in the final proof or not. This is an easy way to set the granularity of the resulting natural language explanation of the proof. Moreover, the priority of each rule in the underlying proof search can be adjusted through a popup-menu. Again, all interface elements are explained by tooltips as soon as the mouse moves over them.

Once the prover is configured, the proof task is ready to be submitted. The 'submit'-action collects all settings from the previous actions, in particular the chosen goal and knowledge base, and displays them for a final check. Hitting the 'Prove'-button submits all data to the *Theorema* kernel and automatically proceeds to the 'inspect'-action. Figure 3 (right) displays the corresponding proof tree as it develops during proof generation. The nodes in the proof tree differ in shape, color, and content depending on node type and status. As soon as the proof is finished, some proof information is written back into that notebook, in which the proof goal has been stated. In addition to an indicator of proof success or failure and a summary of settings used at the time of proof generation, this information contains two important buttons:

1. A button to display the proof in natural language in a separate window as shown in Figure 3 (left). This feature is in essence the same as we had it in *Theorema 1.0* [7]. The 'inspect'-tab in the *Theorema* commander and the proof display are connected in both directions: clicking a node in the proof tree jumps to the respective text blocks in the proof display describing the corresponding proof step; clicking a cell in the proof display marks the corresponding tree node with a small

notebook efficiently in an external file, such that they can be loaded quickly into a *Theorema* session. Since this is not a user-interface issue, we do not go into further details here.

[4]With *built-in knowledge* we refer to knowledge built into the *Theorema* language semantics. As an example, '+' is by default an uninterpreted operator. Using some built-in knowledge one can link '+' to the addition of numbers available in the *Theorema* language. This is a feature inherited from *Theorema 1.0*.

black triangle. In combination this offers a nice possibility to navigate through a proof. As one can see from Figure 3, all formula labels used in the natural proof presentation use tooltips to show the full formula, to which they refer.

2. A button to restore all settings in the *Theorema* commander to the values they had at the time of proof generation, which is a quick way to rerun a proof.

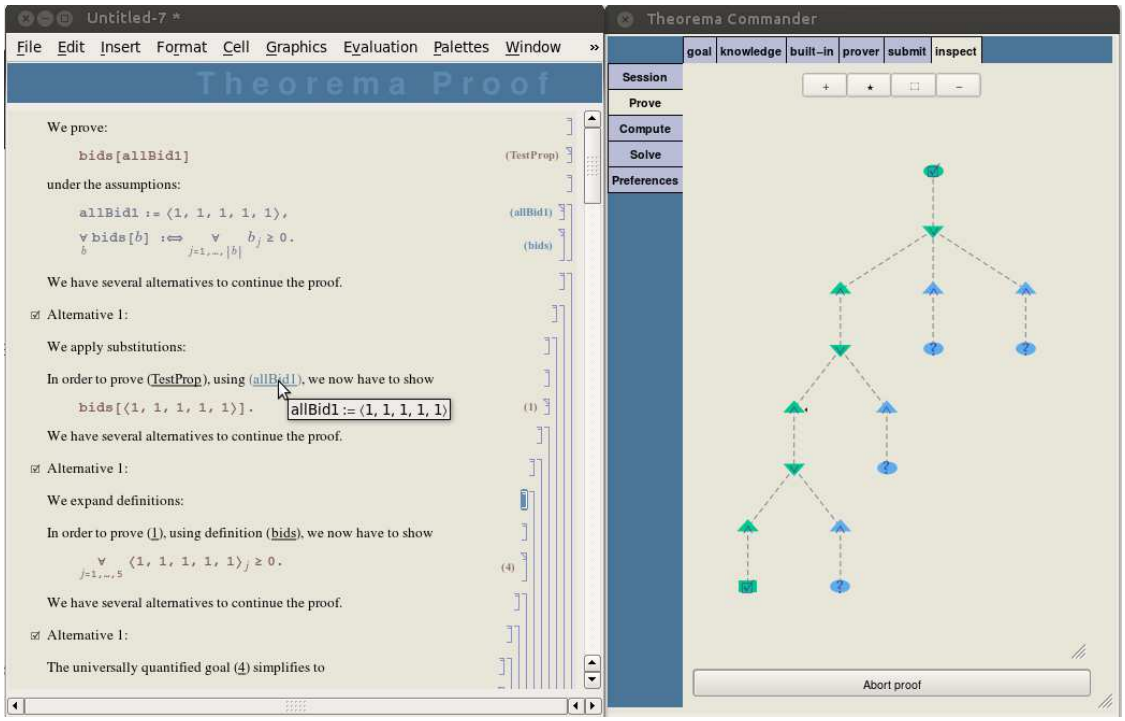


Figure 3: The 'Prove'-activity: a generated proof (left) and the corresponding 'inspect'-action (right).

Other activities The 'Session'-activity consists of structuring formulae into definitions, theorems, etc., arranging global declarations (see Section 3.1), inspecting the session, inputting formulae, and the development and maintenance of knowledge archives. In the 'Compute'-activity, a user sets up the expression to be computed and selects the knowledge base and the built-in knowledge to be used in the computation (using knowledge- and built-in browsers as described for proving above). Knowledge selections for proving are independent from those used for computations.

In the 'Preferences'-activity we collect everything regarding system setup, such as e.g. the preferred language. The entire GUI is language independent in the sense that no single English string (for GUI labels, button labels, explanations, tooltips, etc.) is hardcoded in its implementation, but all strings are constants, whose definitions are collected in several language-setup files. For effective language translation it is important that users have access to the language-setup files so that every user has the possibility to translate the system into her language and that new languages can be integrated with minimal effort. The *Theorema 2.0* architecture is such that the language selection menu in the 'Preferences' will offer the choice among all languages, for which a setup file is present (in a certain directory). This has the effect that, for the translation into a new language, only the English files have to be copied and renamed,

and the English texts need to be translated. Without any further action, the new language can be selected from the menu, and voilà the GUI runs in the new language.

Some other aspects of internationalization are already solved by Mathematica, e.g. the availability of language dependent special characters, unicode, country-specific number formating, etc., others will be considered in future work, e.g. placement of buttons and the "logical direction" of action-wizards for languages written from right to left. In particular for educational purposes that we envisage for *Theorema 2.0*, internationalization is of utmost importance.

An important detail that makes all this possible is the decision to license *Theorema 2.0* under GPL[5]. This gives all users access not only the language-setup files but to the entire source code. An attractive perspective for user contribution to the system could then also be the development of new inference rules or proof strategies. These are just Mathematica programs, and there is a rich library of *Theorema* programs that is ready for use in the implementation of inference rules and strategies.

3.3 The Virtual Keyboard

The last component to be described briefly is the *virtual keyboard*, see the screenshot in Figure 1. Although much of the typical input can be given through buttons and palettes, it turns out that still the keyboard is the most efficient way to enter expressions, at least once a user is a little familiar with the system. Therefore, the *Theorema*-stylesheets define keyboard shortcuts for the most frequently used *Theorema* expressions. In the absence of a physical keyboard—e.g. when working on a tablet computer or on an interactive whiteboard in an educational context—we provide the virtual keyboard, which is an arrangement of buttons imitating a physical keyboard. It consists of a character block for the usual letters and a numeric keypad (numpad) for digits and common arithmetic operators like on common keyboards. As a generalization of the numpad, we provide a *sympad* (to the far right) and an *expad* (to the left) for common mathematical symbols and expressions, respectively. Using modifier keys like Shift, Mod, Ctrl and more, every key on the board can be equipped with many different meanings depending on the setting of the modifiers. We believe that the virtual keyboard is a very powerful input component for mathematical expressions, which will prove useful even in the presence of a physical keyboard, where the buttons react to mouse-clicks.

4 Conclusion

Some of the features are implemented currently as 'proof of concept' and need to be completed in the near future to get a system that can be used for case studies. As an example, the *Theorema* language syntax, from parsing via formatted output to computational semantics, is only implemented for a fraction of what we already had in *Theorema 1.0*. Due to the fact that the already implemented parts are the most complicated ones and that we paid a lot of attention to a generic programming style, we are optimistic that progress can be made quickly in that direction.

The bigger part of the work to be done is the re-implementation of all provers that we already had in *Theorema 1.0*. What we already have now is the generic proof search procedure and the mechanism of inference rule lists and strategies with their interplay. Two sample strategies, one that models more or less the strategy used in *Theorema 1.0* and another one that does a more fine-grained branching on alternative inference rules being applicable, are already available, but no report on their performance can be given at

[5]The system will be available from GITHUB by mid-July 2013, the *Theorema* homepage `http://www.risc.jku.at/research/theorema/software/` will provide more information from then on.

this stage. The big effort is now to provide all the inference rules for standard predicate logic including all the extensions that the *Theorema* language supports. As soon as this is completed we can engage in case studies trying out the system in some real-world theory formalization and in education, for which we plan a hybrid interactive-automatic proof strategy to be available. Towards university education in mathematics and logic, we see a big potential for an interactive proof-assistant based on the new user interface, in particular the proof-tree navigation presented in Section 3.2.

References

[1] B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz & W. Windsteiger (2006): *Theorema: Towards Computer-Aided Mathematical Theory Exploration*. *Journal of Applied Logic* 4(4), pp. 470–504, doi:10.1016/j.jal.2005.10.006.

[2] B. Buchberger, C. Dupre, T. Jebelean, F. Kriftner, K. Nakagawa, D. Vasaru & W. Windsteiger (2000): *The Theorema Project: A Progress Report*. In M. Kerber & M. Kohlhase, editors: *Symbolic Computation and Automated Reasoning (Proceedings of CALCULEMUS 2000, Symposium on the Integration of Symbolic Computation and Mechanized Reasoning)*, St. Andrews, Scotland, Copyright: A.K. Peters, Natick, Massachusetts, pp. 98–113.

[3] B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuta & D. Vasaru (1997): *A Survey of the Theorema project*. In W. Kuechlin, editor: *Proceedings of ISSAC'97 (International Symposium on Symbolic and Algebraic Computation, Maui, Hawaii, July 21-23, 1997)*, ACM Press, pp. 384–391, doi:10.1145/258726.258853.

[4] Florian Kammüller, Markus Wenzel & Lawrence C. Paulson (1999): *Locales: A Sectioning Concept for Isabelle*. In: *Theorem Proving in Higher Order Logics (TPHOLs'99), LNCS 1690*, Springer, pp. 149–165, doi:10.1007/3-540-48256-3_11.

[5] Manfred Kerber, Christoph Lange, Colin Rowat & Wolfgang Windsteiger (2013): *Developing an Auction Theory Toolbox*. In Manfred Kerber, Christoph Lange & Colin Rowat, editors: *AISB 2013*. Available at `http://www.cs.bham.ac.uk/research/projects/formare/events/aisb2013/proceedings.php`.

[6] G. Mayrhofer, S. Saminger & W. Windsteiger (2007): *CreaComp: Experimental Formal Mathematics for the Classroom*. In Shangzhi Li, Dongming Wang, & Jing-Zhong Zhang, editors: *Symbolic Computation and Education*, World Scientific Publishing Co., Singapore, New Jersey, pp. 94–114, doi:10.1142/9789812776006_0006.

[7] W. Windsteiger, B. Buchberger & M. Rosenkranz (2006): *Theorema*. In Freek Wiedijk, editor: *The Seventeen Provers of the World*, *LNAI* 3600, Springer Berlin Heidelberg New York, pp. 96–107, doi:10.1007/11542384_14.

[8] S. Wolfram (1996): *The Mathematica Book*, third edition. Wolfram Media/Cambridge University Press.

S. Linton and R. Sebastiani (ed.), Proc. of the Calculemus 2001 Symposium, pp. 28-47.

On a Solution of the Mutilated Checkerboard Problem using the *Theorema* Set Theory Prover

Wolfgang Windsteiger*[1]

[1]*RISC Institute*
A-4232 Hagenberg, Austria
Wolfgang.Windsteiger@RISC.Uni-Linz.ac.at

Abstract

The Mutilated Checkerboard Problem has some tradition as a benchmark problem for automated theorem proving systems. Informally speaking, it states that an 8 by 8 checkerboard with the two opposite corners removed cannot be covered by dominoes. Various solutions using different approaches have been presented since its original statement by John McCarthy in 1964. An elegant four–line proof has been given on paper by McCarthy himself in 1995, which is based on a formulation of the original problem in set theory. Since then, the checkerboard problem stands as a benchmark problem in particular also for automated set theory provers. In this paper, we are going to present a complete proof of the checkerboard problem using the *Theorema Set Theory prover*.

1. Introduction

The Mutilated Checkerboard (british for Chessboard) Problem goes back to John McCarthy (http://www-formal.stanford.edu/jmc), who stated the problem originally in a Stanford AI memo in 1964, see (McCarthy, 1964): An 8 by 8 checkerboard with two diagonally opposite squares removed cannot be covered by dominoes each of which covers two rectilinearly adjacent squares. Different proofs of this statement have been formulated by among others Shmuel Winograd, Marvin Minsky and Dimitri Stefanyuk, none of them published according to McCarthy, and kind of a contest started to formulate the most non-creative proof

*This work has been supported by the "SFB Numerical and Symbolic Scientific Computing" (F013) at the University of Linz, the "PROVE Project" supported by the regional government of Upper Austria, and the european union "CALCULEMUS Project" (HPRN–CT–2000–00102).

of the statement. In 1993, an interactive proof using the Boyer-Moore prover NQTHM was given in (Subramanian, 1993) and (Subramanian, 1996), whereas William McCune gave a proof using MACE to find a finite model, see (McCune, 1995). In 1996, a proof in higher order logic was presented by Andrews and Bishop, see (AndrewsBishop, 1996), using the TPS system. In 1995, McCarthy himself gave a formulation of the original problem in set theory and a four-line proof, see (McCarthy, 1995). From the abstract of his article: While no present system that I know of will accept either the formal description or the proof, I claim that both should be admitted in any heavy duty set theory. At the same conference, where McCarthy presented this challenge, at the QED Workshop II in Warshaw, Grzegorz Bancerek came up with the set theoretic formulation of McCarthy accepted and checked by Mizar, which took about 400 lines, see (Bancerek, 1995).

The key to the four-line proof of McCarthy is the fact that three propositions are thrown in, which are intuitively correct when having the picture of the colored chessboard in one's mind, but, nevertheless, lack a proof in his presentation. What we shall show in the exploration of the mutilated checkerboard in *Theorema* is

- the reformulation of the definitions and the theorem into the language available in *Theorema*,
- the proof of the Theorem *and* the proofs of *all* intermediate propositions,
- a systematic build up of the "theory of dominoes and partial coverings on a chessboard" augmented also by *computations* in order to get a feeling how the new functions and predicates introduced behave on small examples.

The solution shown below is *one* possible solution in *Theorema*, neither necessarily the most elegant one nor the shortest. In particular, the computations at the beginning do not at all contribute to the final solution, but it was the intention to show, how the *Theorema* system would support the entire process of mathematical theory development. As a matter of fact, experiments with newly invented notions, i.e. computations, are most of the time the origin of the development of involved theories and fancy theorems, since some key properties of the new notions can be observed during the experiments or improvements of the computations turn out to be necessary due to tremendous computation times. This process of "every-day work of a mathematician" has been called the *creativity spiral of mathematics* in (Buchberger, 1993). It is an attempt to argue that the *Theorema* system can support the working mathematician throughout the *entire cycle* of this spiral. Moreover, the solution presented in this paper follows the style of theory exploration as explained in (Buchberger, 1999), with several exploration rounds starting from the initial definitions finally arriving at the theorem of interest. The first exploration round in Section 2 introduces the set theoretic formulation of the basic concepts needed in the formulation of

the checkerboard problem and some immediate properties needed later. Exploration round 2 in Section 3 introduces the new notion of "dominoes" and leads to a proof of a lemma corresponding to one of McCarthy's unproved propositions. Round 3 described in Section 4 introduces the new concept of "partial coverings" and explores the interactions with already known concepts with a Lemma corresponding to the second unproved proposition of McCarthy's as a highlight. The final round in Section 4 collects the results in a proof of McCarthy's theorem.

All intermediate propositions that occur during the entire case study have been proven using the *Theorema* Set Theory prover developed in (Windsteiger, 2001) – except for two propositions in Section 2. However, due to space limitations for the final paper, we omit some of the proofs completely and we leave out some routine parts of the remaining proofs, which we will mark by inserting "..." for the left out proof parts. In the presentation of the proofs, we concentrate on particular features of the Set Theory prover, since this case study was originally motivated as a test for this new prover in the *Theorema* system. See (Windsteiger, 2001) for the complete proofs of *all* propositions and also for a detailed introduction into the theoretical foundations of the Set Theory prover.

2. Exploration Round 1: The Board

As a first remark, McCarthy's formulation uses a slightly different language compared to what *Theorema* offers, but the translation into *Theorema* is pretty straight-forward and we will comment on the details when giving our definitions below. Additionally, we allow ourselves to rename certain concepts according to our personal taste regarding the choice of names instead of just copying from McCarthy's model. We can define the set of pairs representing the chessboard elegantly using multiple integer ranges in the set quantifier. Set difference must be denoted by "\" in case built-in knowledge about set difference is desired. Of course, *Theorema* provides the tools for the user to define any new function or predicate using (almost) any desired notion, but if available semantics for notions provided by the *Theorema* language should be applied during computation, or if available inference rules should by applied during proving then the user must stick to the notions fixed inside the *Theorema* system.

Definition["Board",

$$\text{Board} = \{\langle i, j\rangle \mid_{\substack{i=0,\ldots,7\\ j=0,\ldots,7}} \}$$

$$\text{Mutilated-Board} = \text{Board} \setminus \{\langle 0,0\rangle, \langle 7,7\rangle\}$$

]

As indicated in the final proof of the theorem, there is some motivation to define the color of a pair as 0 or 1, where 0 corresponds to a black square and 1 corresponds to a white square on the chessboard. Using the Mod function, which returns the remainder when dividing by an integer, we can easily define the color

as the remainder when dividing the sum of the components by 2. Just think of the components to be the "coordinates on the board".

Definition["Color", any[x],
color[x] = Mod[$x_1 + x_2, 2$]]]

`Compute`[$\{x \mid_{x \in \text{Mutilated-Board}} \text{color}[x] = 0\}$]

$\{\langle 0,2\rangle, \langle 0,4\rangle, \langle 0,6\rangle, \langle 1,1\rangle, \langle 1,3\rangle, \langle 1,5\rangle, \langle 1,7\rangle, \langle 2,0\rangle, \langle 2,2\rangle, \langle 2,4\rangle, \langle 2,6\rangle, \langle 3,1\rangle, \langle 3,3\rangle,$
$\langle 3,5\rangle, \langle 3,7\rangle, \langle 4,0\rangle, \langle 4,2\rangle, \langle 4,4\rangle, \langle 4,6\rangle, \langle 5,1\rangle, \langle 5,3\rangle, \langle 5,5\rangle, \langle 5,7\rangle, \langle 6,0\rangle, \langle 6,2\rangle,$
$\langle 6,4\rangle, \langle 6,6\rangle, \langle 7,1\rangle, \langle 7,3\rangle, \langle 7,5\rangle\}$

`Compute`[$\{x \mid_{x \in \text{Mutilated-Board}} \text{color}[x] = 1\}$]

$\{\langle 0,1\rangle, \langle 0,3\rangle, \langle 0,5\rangle, \langle 0,7\rangle, \langle 1,0\rangle, \langle 1,2\rangle, \langle 1,4\rangle, \langle 1,6\rangle, \langle 2,1\rangle, \langle 2,3\rangle, \langle 2,5\rangle, \langle 2,7\rangle, \langle 3,0\rangle,$
$\langle 3,2\rangle, \langle 3,4\rangle, \langle 3,6\rangle, \langle 4,1\rangle, \langle 4,3\rangle, \langle 4,5\rangle, \langle 4,7\rangle, \langle 5,0\rangle, \langle 5,2\rangle, \langle 5,4\rangle, \langle 5,6\rangle, \langle 6,1\rangle,$
$\langle 6,3\rangle, \langle 6,5\rangle, \langle 6,7\rangle, \langle 7,0\rangle, \langle 7,2\rangle, \langle 7,4\rangle, \langle 7,6\rangle\}$

`Compute`[$|\{x \mid_{x \in \text{Mutilated-Board}} \text{color}[x] = 0\}|$]

30

`Compute`[$|\{x \mid_{x \in \text{Mutilated-Board}} \text{color}[x] = 1\}|$]

32

The key idea for the entire proof is essentially contained in the following observation, which may be conjectured from the previous computations: the mutilated board contains more white squares than black ones. The region covered by dominoes, however, always contains equally many black and white fields. This observation is even independent of the actual size of the chessboard, see also (Subramanian, 1993). For the computation of colors it is sufficient to use the Mathematica built-in function Mod for doing the remainder calculations. However, if we want to *prove* properties of colors, we will need a definition of the Mod function. We use a definition by cases.

Definition["Mod", any[n],

$$\text{Mod}[n,2] := \begin{cases} 0 & \Leftarrow \quad \underset{j \in \mathbb{N}}{\exists}\ n = 2j \\ 1 & \Leftarrow \quad \textit{otherwise} \end{cases} \quad]$$

The Mod function defined in this way has some interesting interaction with the absolute value function on integers. In the spirit of theory exploration, we do not prove these two propositions now, since they would normally have been proven in earlier exploration rounds when introducing the Mod function or the

absolute value function on integers using a specialized prover for this area. (Of course, we could now give a definition of the absolute value function using case distinction, but finally this would be more in the spirit of *proving going back to first principles.*) In our approach of exploring the checkerboard, we assume the Mod function and the absolute value function on integers as *completely explored*, and we assume that the following propositions have been proven in those explorations. In fact, these propositions are *the only propositions* in this case study, for which we will not provide a proof.

Proposition["Mod property even", any$[x, y, z]$,

$$\begin{array}{l}(|x-z|=1) \wedge (\text{Mod}[x+y,2]=0) \Rightarrow (\text{Mod}[z+y,2]=1) \\ (|x-z|=1) \wedge (\text{Mod}[y+x,2]=0) \Rightarrow (\text{Mod}[y+z,2]=1)\end{array} \ ]$$

Proposition["Mod property odd", any$[x, y, z]$,

$$\begin{array}{l}(|x-z|=1) \wedge (\text{Mod}[x+y,2]=1) \Rightarrow (\text{Mod}[z+y,2]=0) \\ (|x-z|=1) \wedge (\text{Mod}[y+x,2]=1) \Rightarrow (\text{Mod}[y+z,2]=0)\end{array} \ ]$$

Starting from the definitions, we can now prove color properties, like for instance the fact:

Proposition["zero or one", any$[x \in \text{Board}]$,

$(\text{color}[x]=0) \vee (\text{color}[x]=1)]$]

`Prove`[Proposition["zero or one"], using→ ⟨Definition["Mod"],Definition["Color"]⟩, built-in→Built-in["Numbers"]]

In the proof of this proposition, we let the prover use built-in knowledge on numbers from the *Theorema* language semantics. Through this facility, we manage to smoothly integrate *computation* into the *Theorema* proving process, because application of built-in semantics knowledge during proving uses the same mechanism that is used by the top-level *Theorema* command Compute.

3. Exploration Round 2: Dominoes

Definition["Domino", any$[x]$,

$$\text{domino-on-board}[x] :\Leftrightarrow \wedge \left\{ \begin{array}{l} x \subseteq \text{Board} \\ |x| = 2 \\ \underset{\substack{x1,x2 \in x \\ x1 \neq x2}}{\forall} \text{adjacent}[x1, x2] \end{array} \right.]$$

Definition["Adjacency", any$[x, y]$,

$\text{adjacent}[x, y] :\Leftrightarrow ((|x_1 - y_1| = 1) \wedge (x_2 = y_2) \vee (|x_2 - y_2| = 1) \wedge (x_1 = y_1))$]

Many interesting properties will be provable when exploring these two new notions. Still, we start doing some computations first.

`Compute`[domino-on-board$[\{\langle 3,3\rangle, \langle 3,4\rangle\}]$]

True

`Compute`[domino-on-board$[\{\langle 3,3\rangle, \langle 3,4\rangle, \langle 3,5\rangle\}]$]

False

A domino is part of the board and its components are adjacent.

Proposition["dominoes adjacent", any$[X]$,

$\text{domino-on-board}[X] \Rightarrow X \subseteq \text{Board} \wedge \underset{\substack{x,y\in X \\ x\neq y}}{\exists} \text{adjacent}[x, y]$]

`Prove`[Proposition["dominoes adjacent"],using→ Definition["Domino"]]

Prove:

(Proposition (dominoes adjacent)) ...,

under the assumption:

(Definition (Domino)) ....

We assume

(1) domino-on-board$[X_0]$,

and show

(2) $X_0 \subseteq \text{Board} \wedge \underset{x,y}{\exists} ((x \in X_0 \wedge y \in X_0 \wedge (x \neq y)) \wedge \text{adjacent}[x, y])$.

We prove the individual conjunctive parts of (2):
Formula (1), by (Definition (Domino)), implies:

(3) $(|X_0| = 2) \wedge \underset{x1,x2}{\forall} (x1 \in X_0 \wedge x2 \in X_0 \wedge (x1 \neq x2) \Rightarrow \text{adjacent}[x1, x2]) \wedge X_0 \subseteq$ Board.

Formula (2.1) is true because it is identical to (3.3).
Formula (1), by (Definition (Domino)), implies:

(4) $(|X_0| = 2) \wedge \underset{x1,x2}{\forall} (x1 \in X_0 \wedge x2 \in X_0 \wedge (x1 \neq x2) \Rightarrow \text{adjacent}[x1, x2]) \wedge X_0 \subseteq$ Board.

From what we already know follows:
From (4.1) we can infer

(5) $X1_0 \in X_0$,

(6) $X1_1 \in X_0$,

(7) $X1_0 \neq X1_1$.

From (4.3) we can infer

(8) $\underset{X2}{\forall} (X2 \in X_0 \Rightarrow X2 \in \text{Board})$.

Now, let $y := X1_0$. Thus, for proving (2.2) it is sufficient to prove:

(12) $\underset{x}{\exists} ((x \in X_0 \wedge X1_0 \in X_0 \wedge (x \neq X1_0)) \wedge \text{adjacent}[x, X1_0])$.

Now, let $x := X1_1$. Thus, for proving (12) it is sufficient to prove:

(68) $(X1_1 \in X_0 \wedge X1_0 \in X_0 \wedge (X1_1 \neq X1_0)) \wedge \text{adjacent}[X1_1, X1_0]$.

Formula (68.1.1) is true because it is identical to (6).
Formula (68.1.2) is true because it is identical to (5).
Proof of (68.1.3) $X1_1 \neq X1_0$:
We prove (68.1.3) by contradiction.
We assume

(69) $X1_1 = X1_0$,

and show *a contradiction*.
Formula (7), by (69), implies:

(72) $X1_0 \neq X1_0$.

Using available computation rules we can simplify the knowledge base:
Formula (72) simplifies to

(73) *False*,

Formula (a contradiction) is true because the assumption (73) is false.
Formula (68.2), using (4.2), is implied by:

(74) $X1_0 \in X_0 \wedge X1_1 \in X_0 \wedge (X1_1 \neq X1_0)$.

Formula (74.1) is true because it is identical to (5).
Formula (74.2) is true because it is identical to (6).
We prove (74.3) by contradiction.
We assume

(75) $X1_1 = X1_0$,

and show *a contradiction*.
Formula (7), by (75), implies:

(78) $X1_0 \neq X1_0$.

Using available computation rules we can simplify the knowledge base:
Formula (78) simplifies to

(79) *False*,

Formula (a contradiction) is true because the assumption (79) is false. □

We want to emphasize the activation of a special inference rule that allows to choose certain elements from a set known to be finite in the previous proof. This rule is applied when deducing formulae (5), (6), and (7) from formula (4.1).

Adjacent fields have opposite colors.

Proposition["adjacent of black are white", any$[x, y]$,
adjacent$[x, y] \wedge$ color$[x] = 0 \Rightarrow$ color$[y] = 1$]]

`Prove`[Proposiiton["adjacent of black are white"],
using $\rightarrow \langle$Proposition["Mod property even"],
Definition["Adjacency"], Definition["Color"]$\rangle$]

See (Windsteiger, 2001) for this proof and also for the proof of the analog proposition for white fields.

Proposition["adjacent of white are black", any$[x, y]$,
adjacent$[x, y] \wedge$ color$[x] = 1 \Rightarrow$ color$[y] = 0$]]

Using these propositions, we can now prove one of the statements from McCarthy's proof. (We need knowledge about built-in numbers in order to be able to do simplification on arithmetic expressions.)

Lemma["different color", any$[X]$,
domino-on-board$[X] \Rightarrow \underset{u,v \in X}{\exists} ((\text{color}[u] = 0) \wedge (\text{color}[v] = 1))$]

`Prove`[Lemma["different color"],
using $\rightarrow \langle$ Proposition["dominoes adjacent"],
Proposition["zero or one"], Proposition["adjacent of black are white"],
Proposition["adjacent of white are black"]$\rangle$,
built-in $\rightarrow$ Built-in["Numbers"]]

Prove:

(Lemma (different color)) ...,

under the assumptions:

(Proposition (dominoes adjacent)) ...,

(Proposition (zero or one)) $\underset{x}{\forall}$ ($x \in$ Board $\Rightarrow$ (color$[x] = 0) \vee$ (color$[x] = 1)$),

(Proposition (adjacent of black are white)) $\underset{x,y}{\forall}$ (adjacent$[x, y] \wedge$ (color$[x] = 0) \Rightarrow$ (color$[y] = 1)$),

(Proposition (adjacent of white are black)) $\underset{x,y}{\forall}$ (adjacent$[x, y] \wedge$ (color$[x] = 1) \Rightarrow$ (color$[y] = 0)$).

Formula (Proposition (dominoes adjacent)) simplifies to

(1) $\underset{X}{\forall}$ domino-on-board$[X] \Rightarrow X \subseteq$ Board $\wedge \underset{x,y}{\exists} (x \in X \wedge y \in X \wedge (x \neq y) \wedge$ adjacent$[x, y])$,

Using available computation rules we evaluate (Lemma (different color)):

(2) $\underset{X}{\forall}$ (domino-on-board$[X] \Rightarrow \underset{u,v}{\exists} (u \in X \wedge v \in X \wedge$ (color$[u] = 0) \wedge$ (color$[v] = 1)))$.

Formula (1) is simplified to:

(3) $\underset{X}{\forall}$ (domino-on-board$[X] \Rightarrow X \subseteq$ Board$) \wedge \underset{X}{\forall} \Big($domino-on-board$[X] \Rightarrow \underset{x,y}{\exists} (x \in X \wedge y \in X \wedge (x \neq y) \wedge$ adjacent$[x, y])\Big)$.

By (3.2), we can take an appropriate Skolem function such that

(4) $\underset{X}{\forall}$ (domino-on-board$[X] \Rightarrow x_0[X] \in X \wedge y_0[X] \in X \wedge (x_0[X] \neq y_0[X]) \wedge$ adjacent$[x_0[X], y_0[X]])$,

We assume

(5) domino-on-board$[X_0]$,

and show

(6) $\underset{u,v}{\exists} (u \in X_0 \wedge v \in X_0 \wedge$ (color$[u] = 0) \wedge$ (color$[v] = 1))$.

Formula (5), by (3.1), implies:

(7) $X_0 \subseteq$ Board.

From (7) we can infer

(8) $\underset{X1}{\forall} (X1 \in X_0 \Rightarrow X1 \in$ Board$)$.

Formula (5), by (4), implies:

(9) adjacent$[x_0[X_0], y_0[X_0]] \wedge x_0[X_0] \in X_0 \wedge y_0[X_0] \in X_0 \wedge (x_0[X_0] \neq y_0[X_0])$.

From (9.2) we can infer

(10) $X_0 \neq \{\}$.

Formula (9.1), by (Proposition (adjacent of black are white)), implies:

(12) (color$[x_0[X_0]] = 0) \Rightarrow$ (color$[y_0[X_0]] = 1)$.

Formula (9.1), by (Proposition (adjacent of white are black)), implies:

(13) $(\text{color}[x_0[X_0]] = 1) \Rightarrow (\text{color}[y_0[X_0]] = 0)$.

Formula (9.2), by (8), implies:

$x_0[X_0] \in \text{Board}$,

which, by (Proposition (zero or one)), implies:

(14) $(\text{color}[x_0[X_0]] = 0) \vee (\text{color}[x_0[X_0]] = 1)$.

Formula (9.3), by (8), implies:

$y_0[X_0] \in \text{Board}$, which, by (Proposition (zero or one)), implies:

(15) $(\text{color}[y_0[X_0]] = 0) \vee (\text{color}[y_0[X_0]] = 1)$.

We prove (6) by case distinction using (15).
Case (15.1) $\text{color}[y_0[X_0]] = 0$:
We prove (6) by case distinction using (14).
Case (14.1) $\text{color}[x_0[X_0]] = 0$:
Formula (14.1), by (12), implies:
$\text{color}[y_0[X_0]] = 1$,
which, by (15.1), implies:

(17) $0 = 1$.

Using available computation rules we can simplify the knowledge base:
Formula (17) simplifies to

(18) *False*,

Formula (6) is true because the assumption (18) is false.
Case (14.2) $\text{color}[x_0[X_0]] = 1$:
Now, let $u := y_0[X_0]$. Thus, for proving (6) it is sufficient to prove:

(20) $\underset{v}{\exists} (y_0[X_0] \in X_0 \wedge v \in X_0 \wedge (\text{color}[y_0[X_0]] = 0) \wedge (\text{color}[v] = 1))$.

Using available computation rules we evaluate (20) using (15.1), (14.2), (10), and (9.4) as additional assumption(s) for simplification:

(24) $\underset{v}{\exists} (y_0[X_0] \in X_0 \wedge v \in X_0 \wedge (\text{color}[v] = 1))$.

Now, let $v := x_0[X_0]$. Thus, for proving (24) it is sufficient to prove:

(25) $y_0[X_0] \in X_0 \wedge x_0[X_0] \in X_0 \wedge (\text{color}[x_0[X_0]] = 1)$.

Using available computation rules we evaluate (25) using (15.1), (14.2), (10), and (9.4) as additional assumption(s) for simplification:

(27) $y_0[X_0] \in X_0 \wedge x_0[X_0] \in X_0$.

We prove the individual conjunctive parts of (27):
Proof of (27.1) $y_0[X_0] \in X_0$:

Formula (27.1) is true because it is identical to (9.3).
Proof of (27.2) $x_0[X_0] \in X_0$:
Formula (27.2) is true because it is identical to (9.2).
Case (15.2) $\text{color}[y_0[X_0]] = 1$:
We prove (6) by case distinction using (14).
Case (14.1) $\text{color}[x_0[X_0]] = 0$:
... (similar to above)
Case (14.2) $\text{color}[x_0[X_0]] = 1$:
... (similar to above) □

We can now study also some interactions between dominoes, adjacency, and colors. From here on, we list the propositions and leave out some of the proofs due to lack of space. For complete proofs of the propositions see (Windsteiger, 2001)].

Two fields in a domino must be either identical or adjacent.

Proposition["same or adjacent in domino", any$[X, x, y]$,
$\text{domino-on-board}[X] \wedge x \in X \wedge y \in X \Rightarrow x = y \vee \text{adjacent}[x, y]]$]

One domino can neither contain two distinct black fields nor two distinct white fields.

Proposition["black in domino unique", any$[X, x, y]$,
$\text{domino-on-board}[X] \wedge x \in X \wedge y \in X \wedge \text{color}[x] = 0 \wedge \text{color}[y] = 0 \Rightarrow x = y$]

Proposition["white in domino unique", any$[X, x, y]$,
$\text{domino-on-board}[X] \wedge x \in X \wedge y \in X \wedge \text{color}[x] = 1 \wedge \text{color}[y] = 1 \Rightarrow x = y$]

4. Exploration Round 3: Partial Coverings

Definition["Partial Covering", any$[z]$,
$\text{partial-covering}[z] :\Leftrightarrow \underset{x \in z}{\forall} \text{domino-on-board}[x] \wedge \underset{\substack{x,y \in z \\ x \neq y}}{\forall} (x \cap y = \emptyset)$]

The picture to have in mind of a partial covering is that of a set of non-overlapping dominoes. Since we have a clear understanding of dominoes from the previous exploration we skip computations with partial coverings. In this exploration round, we are heading towards the second statement in McCarthy's proof, which, sloppily formulated, says that in a partial covering there are as many black fields as there are white fields. Intuitively, one thinks to have a very clear understanding of the notion "cardinality" that captures the "number of elements in a set", in particular if the sets under consideration are "finite sets".

However, the definition of cardinality of sets tells us that in order to prove that two sets have equal cardinality we have to find a bijective mapping from one set into the other. Alternatively, we could implement a special prover for cardinality that bears in it the clear understanding on finite cardinalities in form of special inference rules. In the spirit of the transition of knowledge from the knowledge base to the level of inference rules, see (Windsteiger, 2001), this would happen after having completed the exploration of the cardinality notion in set theory. We have not implemented such a prover yet, thus, we have to construct the bijection. Before we start proving, we again check by computation whether the statement of interest at least holds in some example:

Definition[“pc”,
$pc := \{\{\langle 1,2\rangle, \langle 1,3\rangle\}, \{\langle 2,2\rangle, \langle 2,3\rangle\}, \{\langle 3,3\rangle, \langle 3,4\rangle\},$
$\{\langle 2,1\rangle, \langle 3,1\rangle\}, \{\langle 3,2\rangle, \langle 4,2\rangle\}\}$]

`Compute`[partial-covering[pc],using→ Definition[“pc”]]

True

`Compute`[$\{u \mid_{u \in \bigcup pc} \text{color}[u] = 0\}$, using→ Definition[“pc”]]

$$\{\langle 1,3\rangle, \langle 2,2\rangle, \langle 3,1\rangle, \langle 3,3\rangle, \langle 4,2\rangle\}$$

`Compute`[$\{u \mid_{u \in \bigcup pc} \text{color}[u] = 1\}$, using→ Definition[“pc”]]

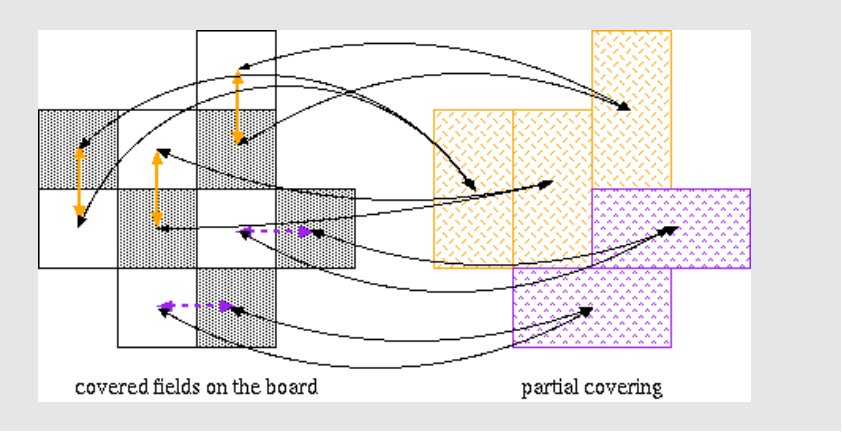
$$\{\langle 1,2\rangle, \langle 2,1\rangle, \langle 2,3\rangle, \langle 3,2\rangle, \langle 3,4\rangle\}$$

`Compute`[$|\{u \mid_{u \in \bigcup pc} \text{color}[u] = 0\}| = |\{u \mid_{u \in \bigcup pc} \text{color}[u] = 1\}|$,
using → Definition[“pc”]]

True

At least in this example, it is really true that there are equally many black fields as white fields in the covering. Note, that this property does certainly not hold for any set of fields on a board, as one can easily verify by a counter-example.

Definition[“no pc”,
$no - pc := \{\{\langle 1,2\rangle, \langle 1,3\rangle\}, \{\langle 2,2\rangle, \langle 2,3\rangle\}, \{\langle 3,3\rangle, \langle 3,4\rangle\},$
$\{\langle 2,1\rangle, \langle 3,1\rangle\}, \{\langle 3,1\rangle, \langle 4,1\rangle\}\}$]



`Compute`[partial-covering[no-pc],using→ Definition[“no pc”]]

False

`Compute`[$\{u \mid_{u \in \cup no-pc}$ color$[u] = 0\}$, using→ Definition[“no pc”]]

$$\{\langle 1,3\rangle, \langle 2,2\rangle, \langle 3,1\rangle, \langle 3,3\rangle\}$$

`Compute`[$\{u \mid_{u \in \cup no-pc}$ color$[u] = 1\}$, using→ Definition[“no pc”]]

$$\{\langle 1,2\rangle, \langle 2,1\rangle, \langle 2,3\rangle, \langle 3,4\rangle, \langle 4,1\rangle\}$$

An observation, which can be made in the computations is, that the number of black fields corresponds to the number of dominoes in the partial covering! The same is true for the number of white fields, which enforces the number of black fields to coincide with the number of white fields. This idea is also depicted in Figure 1.

We therefore investigate the interplay between covered fields on the board and dominoes contained in the partial covering. A first observation is the uniqueness of the domino in the covering, in which a covered field is contained.

Proposition[“covering maps to unique domino”, any[partial-covering[z]], $\underset{u \in \cup z}{\forall} \underset{\substack{x,y \in z \\ u \in x \wedge u \in y}}{\forall} x = y$]

We can even show for each covered field on the board the unique existence of a adomino containing this field. Note that we must formulate the uniqueness

property explicitly by means of predicate logic, because the predicate logic prover does not yet contain special inference rules for the ∃!-quantifier (there exists a unique).

Proposition["covering defines unique domino", any[partial-covering[z]],
$\underset{u\in\cup z}{\forall}\ \underset{x\in z}{\exists}\ u \in x \wedge \text{domino-on-board}[x] \wedge \underset{y\in z}{\forall}\ u \in y \Rightarrow x = y$]

`Prove`[Proposition["covering defines unique domino"],
using → Definition["Partial Covering"]]

Prove:

(Proposition (covering defines unique domino)) ...,

under the assumption:

(Definition (Partial Covering)) ....

...

We assume

(4) $u_0 \in \cup z_0$,

and show

(5) $\underset{x}{\exists}\ x \in z_0 \wedge u_0 \in x \wedge \text{domino-on-board}[x] \wedge \underset{y}{\forall}\ y \in z_0 \Rightarrow u_0 \in y \Rightarrow x = y$.

From (4) we know by definition of the big ∪-operator that we can choose an appropriate value such that

(6) $z1_0 \in z_0$,

(7) $u_0 \in z1_0$.

Now, let $x := z1_0$. Thus, for proving (5) it is sufficient to prove:

(10) $z1_0 \in z_0 \wedge u_0 \in z1_0 \wedge \text{domino-on-board}[z1_0] \wedge \underset{y}{\forall}\ (y \in z_0 \Rightarrow (u_0 \in y \Rightarrow (z1_0 = y)))$.

We prove the individual conjunctive parts of (10):
Formula (10.1) is true because it is identical to (6).
Formula (10.2) is true because it is identical to (7).
Formula (2), by (Definition (Partial Covering)), implies:

(12) $\underset{x}{\forall}\ (x \in z_0 \Rightarrow \text{domino-on-board}[x]) \wedge \underset{x,y}{\forall}\ (x \in z_0 \wedge y \in z_0 \wedge (x \neq y) \Rightarrow (x \cap y = \{\}))$.

Formula (6), by (12.1), implies:

(13) domino-on-board[$z1_0$].

Formula (10.3) is true because it is identical to (13).
Proof of (10.4) $\underset{y}{\forall}\,(y \in z_0 \Rightarrow (u_0 \in y \Rightarrow (z1_0 = y)))$:

We assume

(14) $y_0 \in z_0$,

We assume

(17) $u_0 \in y_0$

and show

(18) $z1_0 = y_0$.

From (17) together with (7) we know

(20) $u_0 \in y_0 \cap z1_0$.

From (20) we can infer

(21) $y_0 \cap z1_0 \neq \{\}$.

Formula (2), by (Definition (Partial Covering)), implies:

(22) $\underset{x}{\forall}\,(x \in z_0 \Rightarrow \text{domino-on-board}[x]) \wedge \underset{x,y}{\forall}\,(x \in z_0 \wedge y \in z_0 \wedge (x \neq y) \Rightarrow (x \cap y = \{\}))$.

Formula (21), by (22.2), implies:

(23) $\neg(y_0 \in z_0 \wedge z1_0 \in z_0 \wedge (y_0 \neq z1_0))$.

Formula (23) is simplified to

(25) $(y_0 \notin z_0) \vee (z1_0 \notin z_0) \vee (y_0 = z1_0)$,

From (14) and (25) we obtain by resolution

(26) $(z1_0 \notin z_0) \vee (y_0 = z1_0)$.

From (6) and (26) we obtain by resolution

(27) $y_0 = z1_0$.

Using available computation rules we evaluate (18) using (27), (21), (19), (8), and (9) as additional assumption(s) for simplification:

$z1_0 = y_0$ ▷

$z1_0 = y_0$ ▷ (by Mathematica's FullSimplify)

True.

(28) *True*.

Formula (28) is true because it is the constant True. □

(Note the final step in the last proof: we could verify $z1_0 = y_0$ *by computation*, because the standard setting of the option "*SimplifyFormula* → *True*" applies

term simplification using additionally certain assumptions from the knowledge base. Essentially, we use the Mathematica built-in FullSimplify for term simplification, which allows also assumptions to be used during simplification, where we pass all number arithmetic related formulae from the knowledge base. In the proof above, this yields the simplification problem $z1_0 = y_0$ under the assumption $y_0 = z1_0$ (from formula (27)), which yields True. In the above example, it would not be necessary to use FullSimplify in order to validate $z1_0 = y_0$ but in other examples it proves useful.) Now, since for all $u \in \cup z$, where z is a partial covering, there exists a *unique* $x \in z$ with $u \in x \wedge$ domino-on-board$[x]$, we may define a function using the "such that" quantifier $\ni$. We will address this domino as "*the* covering domino of u".

Definition["Covering Domino", any$[u,$ partial-covering$[z]]$, with$[u \in \cup z]$
covering-domino$_z[u] := \underset{x \in z}{\ni} u \in x \wedge$ domino-on-board$[x]$]

The domino that contains a covered field is *the* covering domino of just this field.

Proposition["own covering domino", any$[x,$ partial-covering$[z],$ domino-on-board$[d]]$,
$x \in d \wedge d \in z \wedge x \in \cup z \Rightarrow$ covering-domino$_z[x] = d$)]

`Prove`[Proposition["own covering domino"],
using→ ⟨ Proposition["covering maps to unique domino"],
Definition["Covering Domino"] ⟩]

Prove (we only show the parts dealing with the $\ni$-quantifier):
...
Formula (Definition (Covering Domino)) simplifies to

(2) $\underset{u,z}{\forall}$ partial-covering$[z] \wedge u \in \cup z \Rightarrow$ covering-domino$_z[u] := \underset{x}{\ni} u \in x \wedge$ domino-on-board$[x] \wedge x \in z$,

From (2) we can infer by expansion of the "such that"-quantifier

(3) $\underset{u,z}{\forall}$ (partial-covering$[z] \wedge u \in \cup z \Rightarrow u \in$ covering-domino$_z[u] \wedge$ domino-on-board[covering-domino$_z[u]] \wedge$ covering-domino$_z[u] \in z$).

...
Proof of (12.6) covering-domino$_{z_0}[x_0] \in z_0$:
Formula (6.3), by (3), implies:

(15) partial-covering$[z_0] \Rightarrow$ domino-on-board[covering-domino$_{z_0}[x_0]] \wedge$
$x_0 \in$ covering-domino$_{z_0}[x_0] \wedge$ covering-domino$_{z_0}[x_0] \in z_0$.

From (4.1) and (15) we obtain by modus ponens

(16) domino-on-board[covering-domino$_{z_0}[x_0]] \wedge x_0 \in$ covering-domino$_{z_0}[x_0] \wedge$
covering-domino$_{z_0}[x_0] \in z_0$.

Formula (12.6) is true because it is identical to (16.3). □

If two covered black (white respectively) fields have the same covering domino in a partial covering, they must be identical.

Proposition["black and same covering domino:identical", any$[u, v,$ partial-covering$[z]]$,
covering-domino$_z[u] =$ covering-domino$_z[v] \wedge u \in \cup z \wedge v \in \cup z \wedge$
$\text{color}[u] = 0 \wedge \text{color}[v] = 0 \Rightarrow u = v$]

Proposition["white and same covering domino:identical", any$[u, v,$ partial-covering$[z]]$,
covering-domino$_z[u] =$ covering-domino$_z[v] \wedge u \in \cup z \wedge v \in \cup z \wedge$
$\text{color}[u] = 1 \wedge \text{color}[v] = 1 \Rightarrow u = v$]

The covering-domino function is a bijective mapping from the set of covered black fields to the set of dominoes in a partial covering.

Lemma["covering domino bijective from black fields", any[partial-covering$[z]]$,
covering-domino$_z$ :: $\{u \mid_{u \in \cup z} \text{color}[u] = 0\} \overset{bij}{\rightarrow} \{d \mid_{d \in z} \text{domino-on-board}[d]\}$]

`Prove`[Lemma["covering domino bijective from black fields"],
using→ ⟨Proposition["own covering domino"], Lemma["different color"],
Proposition["black and same covering domino: identical"] ⟩]

Prove:

(Lemma (covering domino bijective from black fields)) ...,

under the assumptions:

(Proposition (own covering domino)) ...,

(Lemma (different color)) ...,

(Proposition (black and same covering domino: identical)) ....

Using available computation rules we can simplify the knowledge base: Formula (Lemma (different color)) simplifies to

(1) $\underset{X}{\forall}$ (domino-on-board$[X] \Rightarrow \underset{u,v}{\exists}$ $(u \in X \wedge v \in X \wedge (\text{color}[u] = 0) \wedge (\text{color}[v] = 1)))$,

By (1), we can take an appropriate Skolem function such that

(2) $\underset{X}{\forall}$ (domino-on-board$[X] \Rightarrow u_0[X] \in X \wedge v_0[X] \in X \wedge (\text{color}[u_0[X]] = 0) \wedge (\text{color}[v_0[X]] = 1))$,

We assume

(3) partial-covering$[z_0]$,

and show

(4) covering-domino$_{z_0}$ :: $\{u\underset{u}{|}\ u \in \cup z_0 \wedge (\text{color}[u] = 0)\} \overset{bij}{\rightarrow} \cdot \{d\underset{d}{|}$ domino-on-board$[d] \wedge d \in z_0\}$.

In order to show that covering-domino$_{z_0}$ in (4) is bijective, we have to prove:
Injectivity of covering-domino$_{z_0}$:

(5) covering-domino$_{z_0}$ :: $\{u\underset{u}{|}\ u \in \cup z_0 \wedge (\text{color}[u] = 0)\} \overset{inj}{\rightarrow} \cdot \{d\underset{d}{|}$ domino-on-board$[d] \wedge d \in z_0\}$.

In order to show injectivity of covering-domino$_{z_0}$ in (5) we assume

(7) $u3_0 \in \{u\underset{u}{|}\ u \in \cup z_0 \wedge (\text{color}[u] = 0)\}$,

(8) $u4_0 \in \{u\underset{u}{|}\ u \in \cup z_0 \wedge (\text{color}[u] = 0)\}$,

(9) covering-domino$_{z_0}[u3_0]$ = covering-domino$_{z_0}[u4_0]$.

and show

(10) $u3_0 = u4_0$.

From what we already know follows:
From (7) we can infer

(11) $u3_0 \in \cup z_0 \wedge (\text{color}[u3_0] = 0)$.

From (8) we can infer

(12) $u4_0 \in \cup z_0 \wedge (\text{color}[u4_0] = 0)$.

Formula (10), using (Proposition (black and same covering domino: identical)), is implied by:

(17) $\underset{z}{\exists}$ (partial-covering$[z] \wedge u3_0 \in \cup z \wedge u4_0 \in \cup z \wedge (\text{color}[u3_0] = 0) \wedge (\text{color}[u4_0] = 0) \wedge$ (covering-domino$_z[u3_0]$ = covering-domino$_z[u4_0]$)).

Using available computation rules we evaluate (17) using (11.2), (12.2), and (9) as additional assumption(s) for simplification:

(18) $\underset{z}{\exists}$ (partial-covering$[z] \wedge u3_0 \in \cup z \wedge u4_0 \in \cup z \wedge$ (covering-domino$_z[u3_0]$ = covering-domino$_z[u4_0]$)).

Now, let $z := z_0$. Thus, for proving (18) it is sufficient to prove:

(19) partial-covering$[z_0] \wedge u3_0 \in \cup z_0 \wedge u4_0 \in \cup z_0 \wedge$ (covering-domino$_{z_0}[u3_0]$ = covering-domino$_{z_0}[u4_0]$).

Using available computation rules we evaluate (19) using (11.2), (12.2), and (9) as additional assumption(s) for simplification:

(20) partial-covering$[z_0] \wedge u3_0 \in \cup z_0 \wedge u4_0 \in \cup z_0$.

We prove the individual conjunctive parts of (20):
Formula (20.1) is true because it is identical to (3).
Formula (20.2) is true because it is identical to (11.1).
Formula (20.3) is true because it is identical to (12.1).
Surjectivity of covering-domino$_{z_0}$:

(6) covering-domino$_{z_0}$ :: $\{u\big|_u\ u \in \cup z_0 \wedge (\text{color}[u] = 0)\} \overset{surj.}{\rightarrow} \cdot \{d\big|_d$ domino-on-board$[d] \wedge d \in z_0\}$.

In order to show surjectivity of covering-domino$_{z_0}$ in (6) we assume

(21) $d1_0 \in \{d\big|_d$ domino-on-board$[d] \wedge d \in z_0\}$,

and show

(22) $\underset{u5}{\exists}\ u5 \in \{u\big|_u\ u \in \cup z_0 \wedge (\text{color}[u] = 0)\} \wedge (\text{covering-domino}_{z_0}[u5] = d1_0)$.

From what we already know follows:
From (21) we can infer

(23) domino-on-board$[d1_0] \wedge d1_0 \in z_0$.

In order to prove (22) we have to show:

(24) $\underset{u5}{\exists}\ ((\ u5 \in \cup z_0 \wedge (\text{color}[u5] = 0)) \wedge (\text{covering-domino}_{z_0}[u5] = d1_0))$.

Using available computation rules we evaluate (24):

(25) $\underset{u5}{\exists}\ (\ u5 \in \cup z_0 \wedge (\text{color}[u5] = 0) \wedge (\text{covering-domino}_{z_0}[u5] = d1_0))$.

Formula (23.1), by (2), implies:

(26) $u_0[d1_0] \in d1_0 \wedge v_0[d1_0] \in d1_0 \wedge (\text{color}[u_0[d1_0]] = 0) \wedge (\text{color}[v_0[d1_0]] = 1)$.

Now, let $u5 := u_0[d1_0]$. Thus, for proving (25) it is sufficient to prove:

(27) $u_0[d1_0] \in \cup z_0 \wedge (\text{color}[u_0[d1_0]] = 0) \wedge (\text{covering-domino}_{z_0}[u_0[d1_0]] = d1_0)$.

Using available computation rules we evaluate (27) using (26.3)and (26.4) as additional assumption(s) for simplification:

(28) $u_0[d1_0] \in \cup z_0 \wedge (\text{covering-domino}_{z_0}[u_0[d1_0]] = d1_0)$.

We prove the individual conjunctive parts of (28):
In order to show (28.1) we have to show

(29) $\underset{z3}{\exists}\ (u_0[d1_0] \in z3 \wedge z3 \in z_0)$.

Now, let $z3 := d1_0$. Thus, for proving (29) it is sufficient to prove:

(30) $u_0[d1_0] \in d1_0 \wedge d1_0 \in z_0$.

We prove the individual conjunctive parts of (30):
Formula (30.1) is true because it is identical to (26.1).

Formula (30.2) is true because it is identical to (23.2).
Formula (28.2), using (Proposition (own covering domino)), is implied by:

(31) domino-on-board$[d1_0] \wedge$ partial-covering$[z_0] \wedge d1_0 \in z_0 \wedge u_0[d1_0] \in d1_0 \wedge u_0[d1_0] \in \cup z_0$.

We prove the individual conjunctive parts of (31):
Formula (31.1) is true because it is identical to (23.1).
Formula (31.2) is true because it is identical to (3).
Formula (31.3) is true because it is identical to (23.2).
Formula (31.4) is true because it is identical to (26.1). In order to show (31.5) we have to show

(32) $\underset{z4}{\exists} (u_0[d1_0] \in z4 \wedge z4 \in z_0)$.

Now, let $z4 := d1_0$. Thus, for proving (32) it is sufficient to prove:

(33) $u_0[d1_0] \in d1_0 \wedge d1_0 \in z_0$.

We prove the individual conjunctive parts of (33):
Formula (33.1) is true because it is identical to (26.1).
Formula (33.2) is true because it is identical to (23.2). □

This, of course, was the key step for finally proving equal cardinality of the set of white fields and the set of black fields in a partial covering, because an analog proof yields a bijection from the white covered fields to the set of dominoes, which proves equal cardinality of black and white covered fields.

Corollary["covering domino bijective from white fields", any[partial-covering[z]],
covering-domino$_z$ :: $\{u \underset{u \in \cup z}{|} \text{color}[u] = 1\} \overset{bij}{\rightarrow} \cdot \{d \underset{d \in z}{|} \text{domino-on-board}[d]\}$]

There are as many black covered fields on the board as there are dominoes in the partial covering.

Corollary["number of dominoes", any[partial-covering[z]],
$|\{u \underset{u \in \cup z}{|} \text{color}[u] = 0\}| = |\{d \underset{d \in z}{|} \text{domino-on-board}[d]\}|$
$|\{u \underset{u \in \cup z}{|} \text{color}[u] = 1\}| = |\{d \underset{d \in z}{|} \text{domino-on-board}[d]\}|$]

Lemma["equally many black and white covered fields", any[any[z]],
partial-covering$[z] \Rightarrow |\{u \underset{u \in \cup z}{|} \text{color}[u] = 0\}| = |\{u \underset{u \in \cup z}{|} \text{color}[u] = 1\}|$]

This lemma corresponds to the second unproved statement of McCarthy's four-line proof. See (Windsteiger, 2001) for the complete proofs.

Exploration Round 4: The Final Theorem

Now we can go for the final theorem, which, after having explored dominoes, colors, and partial coverings extensively, follows rather easily from the Lemma on the number of black and white covered fields proven at the end of the previous exploration round.

The mutilated checkerboard cannot be covered by dominoes.

Theorem[“mutilated checkerboard”,
$\neg \underset{z}{\exists}$ (partial-covering$[z] \wedge \bigcup z$ = Mutilated-Board)]

`Prove`[Theorem[“mutilated checkerboard”],
using → ⟨Lemma[“equally many black and white covered fields”]⟩,
built-in → ⟨ Built-in[“Tuples”][Subscript, AngleBracket], Built-in[“Numbers”],
Definition[“Board”], Definition[“color”]⟩]

Prove:

(Theorem (mutilated checkerboard)) $\neg \underset{z}{\exists}$ (partial-covering$[z] \wedge \bigcup z$ = Mutilated-Board),

under the assumption:

(Lemma (equally many black and white covered fields)) ....

We prove (Theorem (mutilated checkerboard)) by contradiction. We assume

(1) $\underset{z}{\exists}$ (partial-covering$[z] \wedge \cup z$ = Mutilated-Board),

and show *a contradiction*. Formula (1) simplifies to

(2) $\underset{z}{\exists}$ (partial-covering$[z] \wedge (\cup z = \{\langle 0,1\rangle, \langle 0,2\rangle, << 58 >>, \langle 7,5\rangle, \langle 7,6\rangle\}))$,

By (2) we can take appropriate values such that:

(3) partial-covering$[z_0] \wedge (\cup z_0 = \{\langle 0,1\rangle, \langle 0,2\rangle, << 58 >>, \langle 7,5\rangle, \langle 7,6\rangle\})$.

Formula (3.1), by (Lemma (equally many black and white covered fields)), implies:

$|\{u \mid_{u \in \cup z_0} \text{color}[u] = 0\}| = |\{u \mid_{u \in \cup z_0} \text{color}[u] = 1\}|$, which, by (3.2), implies:

$$(16) \quad \begin{array}{l} |\{u \mid_{u \in \cup\{\langle 0,1\rangle, \langle 0,2\rangle, << 58 >>, \langle 7,5\rangle, \langle 7,6\rangle\}} \text{color}[u] = 0\}| = \\ \qquad |\{u \mid_{u \in \cup\{\langle 0,1\rangle, \langle 0,2\rangle, << 58 >>, \langle 7,5\rangle, \langle 7,6\rangle\}} \text{color}[u] = 1\}| \end{array} .$$

Formula (16) simplifies to

(17) *False*,

Formula (a contradiction) is true because the assumption (17) is false. □

Conclusion

We presented a complete exploration of the checkerboard, dominoes, and partial coverings on a board, finally leading to a complete solution of the Mutilated Checkerboard Problem of McCarthy. This exploration is at the same time a nice demonstration of cooperation of proving and computing, which shows a particular strength of the *Theorema* system, namely the uniform language of expressions that allows proving and computing in the same logical frame without having need of translation between different representations suited for proving or computing. It thereby proves both

- the suitability of *Theorema* as a system supporting the entire cycle of mathematical research work and
- the power of the *Theorema* Set Theory prover as a specialized tool for tackling problems formulated in set theory.

In particular, we want to point out the two-fold intergration between proving and computing in the *Theorema* system. On the one hand, proving and computing are integrated by offering the user commands Prove and Compute embedded in a coherent language frame. On the other hand, the Set Theory prover incorporates *computation* as an *inference step* for finite language constructs, like for instance the finite set representing the mutilated checkerboard in the proof of the main theorem above. The computation in the proof of the theorem can of course only be performed for a given size of the checkerboard, in the above case 8 by 8. For arbitrary size, the final proof cannot be done by computation but it would require some reasoning on *cardinalities* of (finite) sets. This place is, however, the only place in the entire exploration, where the concrete size of the checkerboard is used in a proof, thus, generalization to the n by n case would require only providing additional knowledge on finite cardinalities in order to show that the number of white fields can't be equal to the number of black fields on the mutilated board.

References

P. Andrews and M. Bishop. On Sets, Types, Fixed Points, and Checkerboards. In Pierangelo Miglioli and Ugo Moscato and Daniele Mundici and Mario Ornaghi, editor, *Theorem Proving with Analytic Tableaux and Related Methods. 5th International Workshop, TABLEAUX '96, Palermo*, number 1071 in LNAI, pages 1-15, Springer, May 1996.

G. Bancerek. The Mutilated Chessboard Problem - checked by MIZAR. In Roman Matuszewski, editor, *The QED Workshop II*, pages 25-26, Warshaw University. 1995. Technical Report L/1/95.

B. Buchberger. Mathematica: A System For Doing Mathematics by Computer?.

In A. Miola and M. Temperini, editor, *Advances in the Design of Symbolic Computation Systems*, pages 2-29, Springer Verlag, Wien - New York, 1997. Invited Talk at DISCO 93, Gmunden, Austria.

B. Buchberger. Theory Exploration Versus Theorem Proving. In A. Armando and T. Jebelean, editor, *Electronic Notes in Theoretical Computer Science*, volume 23-3 of pages 67-69, Elsevier, 1999. CALCULEMUS Workshop, University of Trento, Trento, Italy.

J. McCarthy. A Tough Nut for Proof Procedures. Stanford AI project memo, 1964.

J. McCarthy. The Mutilated Checkerboard in Set Theory. In Roman Matuszewski, editor, *The QED Workshop II*, pages 25-26, Warshaw University. 1995. Technical Report L/1/95.

W. McCune. Another Crack in a Tough Nut. *Association for Automated Reasoning Newsletter*, (31):1-3, 1995.

S. Subramanian. *A Mechanized Framework for Specifying Problem Domains and Verifying Plans*. University of Austin, Texas, 1993.

S. Subramanian. An interactive solution to the n x n mutilated checkerboard problem. *Journal of Logic and Computation*, 6(4):573-598, August 1996.

Wolfgang Windsteiger. *A Set Theory Prover in Theorema: Implementation and Practical Applications*. RISC Institute, 2001.

Exploring an Algorithm for Polynomial Interpolation in the *Theorema* System

Wolfgang Windsteiger[*]

RISC Institute, University of Linz
A-4232 Hagenberg, Austria

Abstract

We present a case study using the *Theorema* system to explore an algorithm for polynomial interpolation. The emphasis of the case study lies on formulating mathematical knowledge in *one* language that appears in its syntax close to common mathematical language but is precise enough to formulate all details necessary for proving. Moreover, the language allows the computation of concrete examples without any further translation into an executable language.

---

* This work has been supported by the "Spezialforschungsbereich for Numerical and Symbolic Scientific Computing" (SFB F013) at the University of Linz and the european union "CALCULEMUS Project" (HPRN-CT-2000-00102). The author would like to thank B.Buchberger and M.Rosenkranz for many valuable discussions during the case study that led to this paper.

---

1 Introduction

Existing mathematical software systems (e.g. Mathematica, MAPLE, Gap etc.) have made big progress over the past decades by providing the users with comprehensive libraries of sophisticated algorithms in various areas of mathematics. In parallel, basically independently, the past decades have also produced enormous progress in the automation of the proving activity of mathematicians. These approaches, however, have put their emphasis mainly on proving isolated theorems, where systems like Otter, Spass, or Vampire are rather successful.

The challenge for the future is the theoretical foundation, the design, and implementation of *integral software systems* that guide, support, and at least partially automate the entire process of inventing, proving, and applying mathematical knowledge using mathematical knowledge and method libraries and, as a result, expanding these libraries by the result of this process. Recently, this integral view and research program for the next generation of mathematical software systems has been named "Mathematical Knowledge Management" (MKM) in the first international workshop on MKM, Sept. 14-16, 2001, organized at RISC-Linz by B. Buchberger, see [MKM 03].

From the very beginning the *Theorema* project was meant to give a logical and software technological frame for the entire mathematical knowledge management as an integral coherent process with the following key design objectives and ideas:

- The three main activities of mathematics—proving, computing, and solving—should all be available in one logical and software technological frame. Moreover, the natural interplay between proving, computing, and solving should be supported by the system.

- Domains, functors and categories as a natural and powerful structuring mechanism for generic build-up of systematic knowledge and methods.
- Preference to special proving, simplifying, and solving algorithms for special mathematical theories as opposed to a one-method-approach to proving all of mathematics (e.g. resolution method) with the possibility to link powerful and provenly correct algebraic algorithms (e.g. Cylindrical Algebraic Decomposition for quantifier elimination, Gröbner bases method for systems of algebraic equations, Risch's algorithm for symbolic integration, or Zeilberger's algorithm for sum identities).
- High usability and attractiveness for the working mathematician by using various software technological advances, e.g. flexible syntax imitating the usual textbook-style, proof presentation with high readability and postprocessing capabilities.
- Knowledge management tools for the construction, maintenance, and modification of large mathematical knowledge bases together with system support for integral theory exploration: failure analysis for proofs and conjecture generation based on failure analysis; build-up of libraries of problem types, knowledge types and algorithm types and their systematic use in the theory exploration process.

In this paper, we would like to demonstrate part of the currently available features in *Theorema* in a case study, namely polynomial interpolation. In this case study, we show: 1) How the domain of univariate polynomials can be built up in generic form using the functor construct available in *Theorema*. 2) We demonstrate how the problem of interpolation can be specified in this setting. 3) How a special solution of the interpolation problem, namely by the Neville algorithm, can be formulated. 4) How its correctness proof can be given. 5) How the algorithm can finally be applied to concrete problems.
With this case study we want to demonstrate the following aspects, which play a crucial role in an integral view of the mathematical knowledge management process: 1) Problem specification, algorithmic formulation, correctness proof, and computation (application to concrete examples) can be done within the one and uniform logic and system frame of *Theorema*. 2) The possibility for formulating mathematical knowledge and methods in a generic way that guarantees applicability and re-usability in a wide range of (hierarchically built-up) domains. 3) Attractive choice of syntax, which is close to the common usage of notation in mathematical textbooks and at the same time is formally rigorous in the sense that all formulae (including the algorithms) are just formulae in the underlying predicate logic. We want to particularly emphasize the didactic challenge in this context: on the one hand, every detail must be spelled out unambiguously whereas, on the other hand, we want to stay close to common use (and often ab-use) of mathematical notation used in mathematical texts.
In this paper, we do not yet talk about possibilities in *Theorema* for guiding and supporting the invention process. However, note for example, that proofs for elementary properties of e.g. polynomial evaluation, which are needed for polynomial interpolation, are naturally suggested by the structure of the polynomial domain as defined in the polynomial functor. However, we would also like to emphasize that systematic methods for mathematical exploration, in our view, are not only a desirable goal for completely automating the invention process (which will never be possible by the inherent incompleteness of mathematics) but are a very reasonable and worthwhile research goal for improving the didactics and heuristics of mathematics. In concrete terms this means that at certain stages in the invention process instead of getting support from the system the user may also interact with the system by allowing the user to guide the prover or suggesting the prover the general structure of an algorithm.
This case study is taken from lecture notes used in courses, whose goal is to present the entire content of the first year of mathematics study in an algorithmic fashion. The *Theorema* language turns out to provide a suitable frame for these courses, because the entire mathematical knowledge including all algorithms can be formulated in a style, which later allows *proving* all the properties in subsequent courses. Similar case studies have been initiated for other topics such as Gaussian elimination or Gröbner bases theory.

2 The Polynomial Functor

We present a case study in the domain of univariate polynomials over a field K. Polynomials can be defined to be infinite K-sequences with only finitely many non-zero elements, i.e. the (infinite) direct sum of (infinitely many copies of) the coefficient field. Thus, for each such sequence there must be an index, such that the sequence consists of only zeroes after this index. This is an appropriate setting for a computer-representation of polynomials, since it allows to naturally represent a polynomial by a K-tuple of its coefficients up to the last non-zero entry in the sequence.

In the *Theorema* language, the domain of univariate polynomials over a coefficient field K can be introduced nicely by a *Functor*. Functors are a well-known concept (e.g. in category theory) and the hierachical construction of mathematical domains by functors has already been used in Computer Algebra systems (the use of domains and categories is one of the distinctive design features of the well-known AXIOM system, see [AXIOM]). The algorithmic nature of functors as introduced in the *Theorema* system (see [Buchberger 96a], [Buchberger 96b], and [Windsteiger 99]) relates to how functors are available in the programming language ML. In general, a functor allows to construct a new domain from an already existing domain. In the concrete case, we assume a domain K and construct the domain of polynomials over K, named Poly[K], by defining the characteristic property for the elements in Poly[K] and by defining operations in Poly[K] based on available operations in the underlying domain K. Note that we will present here only part of the functor, namely just those definitions that are relevant for further discussion on the polynomial interpolation algorithm presented in Section 4. The functor definition shown in Figure 1 must be read as follows: The domain Poly[K] is such a domain P, where, for any p, q, n, a, the following operations are defined:

- $\underset{P}{\in}[p]$ (p is an element in P) iff p is a tuple of positive length with elements from K.
- $\underset{P}{x}$ (a new constant x in P) is the tuple $\langle \underset{K}{0}, \underset{K}{1} \rangle$.
- $\deg_P[p]$ (the degree of p in P) is either 0 or it is such an i between 1 and $|p|$ such that … . (The "such a-quantifier" $\underset{i=1,\ldots,|p|}{\ni}$ … is a special language construct available in the *Theorema* language, which stands for "such an i between 1 and $|p|$ satisfying the property …". It provides a formal frame for giving *implicit function definitions.*)
- etc.

The constant x in the polynomial domain plays exactly the role of the "polynomial indeterminate" x when thinking of polynomials as "arithmetic terms" of the form $\sum_{k=0}^{n} p_i\, x^i$. In the functor notation all function, predicate, and object constants carry the domain, for which they are defined, as an underscript, e.g. $\underset{\text{Poly}[K]}{-}$ for subtraction in the domain of polynomials as opposed to $\underset{K}{-}$ for subtraction in the domain K. In the remainder of this paper, all text in gray boxes is *Theorema* input or output as it appears in a *Theorema* session. The syntax used—including all special symbols and typesetting facilities—is machine-readable and the *Theorema* parser translates it unambiguously into *Theorema*'s internal representation.

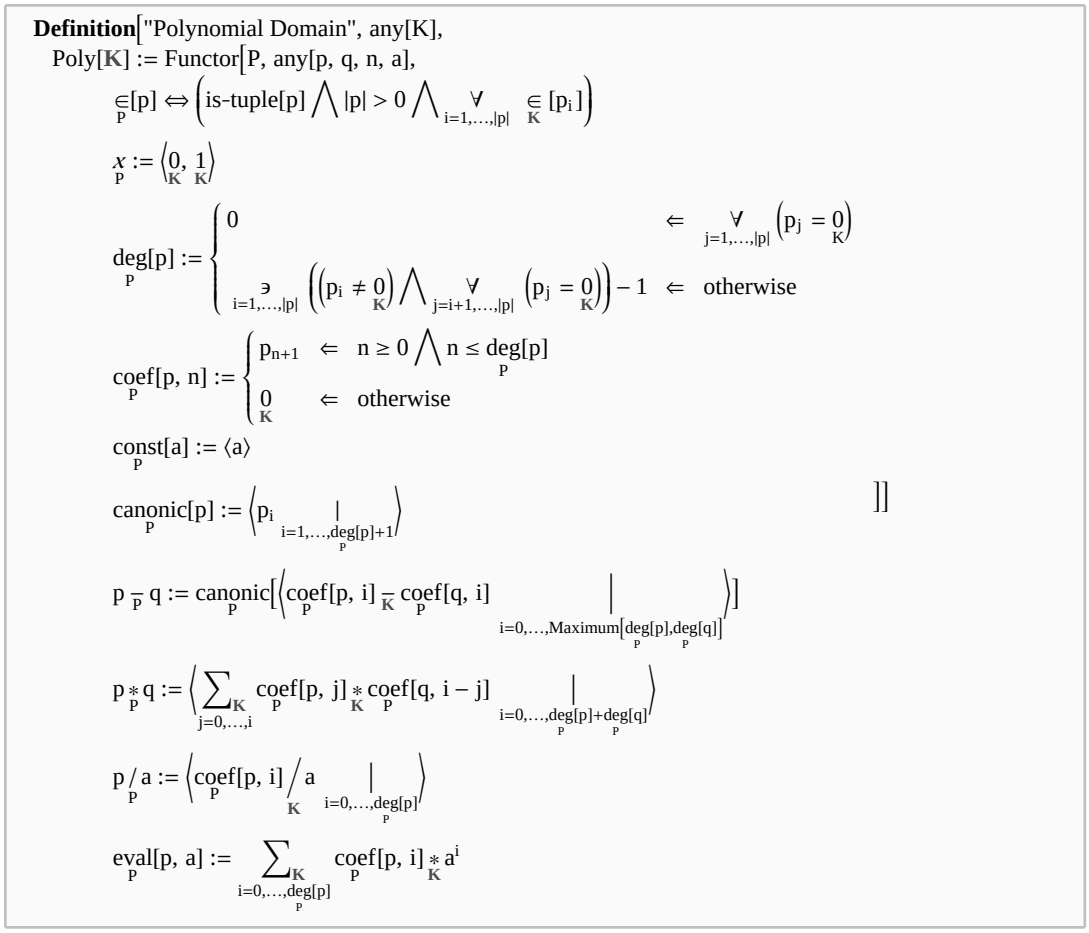
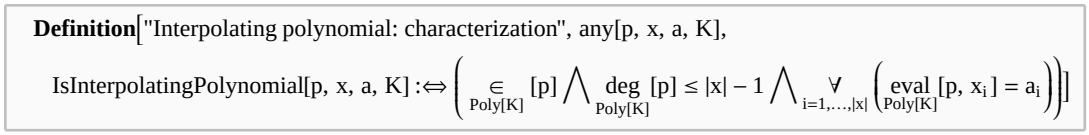


Figure 1: The functor defining the domain of univariate polynomials.

3 Problem Specification: Polynomial Interpolation

Given a polynomial p over K and two tuples x and a, one might ask, whether p evaluates (in Poly[K]) to a_i at x_i (for all $i = 1, \ldots, |x|$), i.e. whether p is an interpolating polynomial for x and a in Poly[K]. This consideration is natural because then the "polynomial function associated with p" would run through all the given points $\langle x_i, a_i \rangle$, which is a crucial property for many applications in mathematics (e.g. several methods for solving equations are based on iteratively solving equations for interpolating polynomials). In *Theorema*, this property can be expressed as follows:



Under certain restrictions—the tuples x and a must be non-empty and have equal length and the elements of x must be mutually distinct—it can be shown that for given x, a, and K there exists a unique polynomial p over K of degree less equal $|x| - 1$ such that IsInterpolatingPolynomial[p, x, a, K]. Of course, it is then desirable to come up with an *algorithm* that computes the interpolating polynomial for given tuples x, a and coefficient field K.

4 Solution to the Interpolation Problem: Neville Algorithm

An ad-hoc solution for an interpolation algorithm can immediately be extracted from the proof of unique existence of the interpolating polynomial. The proof of this fact can be reduced to prove solvability of a system of linear equations, which is always guaranteed under the given restrictions on x and a. The interpolating polynomial can then be computed by solving the linear system. However, there are better algorithms for finding the interpolating polynomial, for instance the *Neville algorithm*, which proceeds by *recursion* over the tuples x and a. Written in *Theorema* the algorithm is given as follows:

Algorithm["Neville", any[x, a, x0, $\overline{x}$, xn, a0, $\overline{a}$, an, K],

NevillePolynomial[⟨x⟩, ⟨a⟩, K] = $\underset{Poly[K]}{const}$[a]

NevillePolynomial[⟨x0, $\overline{x}$, xn⟩, ⟨a0, $\overline{a}$, an⟩, K] =

$$\left(\left(x \underset{Poly[K]}{-} \underset{Poly[K]}{const}[x0]\right) \underset{Poly[K]}{*} \text{NevillePolynomial}[\langle \overline{x}, xn\rangle, \langle \overline{a}, an\rangle, K] \underset{Poly[K]}{-} \right.$$

$$\left. \left(x \underset{Poly[K]}{-} \underset{Poly[K]}{const}[xn]\right) \underset{Poly[K]}{*} \text{NevillePolynomial}[\langle x0, \overline{x}\rangle, \langle a0, \overline{a}\rangle, K]\right) \underset{Poly[K]}{/} \left(xn \underset{K}{-} x0\right)$$

5 Correctness of the Algorithm

The correctness theorem for the Neville algorithm written in *Theorema* syntax is as follows:

Theorem["Neville polynomial is interpolating polynomial", any[is-tuple[x], a, K], with[|x| > 0 ∧ |a| = |x|]
IsInterpolatingPolynomial[NevillePolynomial[x, a, K], x, a, K]]

For automatically proving a formula *for all tuples* x, we can use the *tuple induction prover* available in the *Theorema* system. This prover implements a special prove technique available for tuples, namely Noetherian induction. In order to call this prover, we issue the *Theorema* command

Prove[Theorem["Neville polynomial is interpolating polynomial"], using → KB, by → TupleInduction],

where KB contains the knowledge base of auxiliary assumptions needed for the proof. We will refer to required knowledge from KB in the proof later. The tuple induction prover comes up with a successful and complete proof. Due to space limitations, we show only the key steps of this proof (text in boxes contains explanation of the prove techniques applied, *all the rest*—including formula labels, references, and intermediate text—is generated completely automatically by the prover).

Since x is a tuple an induction over x is set up. Since nothing is known about a and K the prover chooses a, K arbitrary but fixed.

Induction base: $x = \langle x1\rangle$ for arbitrary but fixed x1. We have to show:

(1) |a| = 1 ⇒ IsInterpolatingPolynomial[NevillePolynomial[⟨x1⟩, a, K], ⟨x1⟩, a, K],

We assume

(2) |a| = 1,

and show

(3) IsInterpolatingPolynomial[NevillePolynomial[⟨x1⟩, a, K], ⟨x1⟩, a, K].

From (2), we can infer:

(4) a = ⟨a1⟩,

for some new constant a1.

Formula (3), using (4) and (Algorithm (Neville)), is implied by:

(5) IsInterpolatingPolynomial[$\underset{Poly[K]}{const}$[a1], ⟨x1⟩, ⟨a1⟩, K],

which, using (Definition (Polynomial Domain)), is implied by:

(6) IsInterpolatingPolynomial[⟨a1⟩, ⟨x1⟩, ⟨a1⟩, K],

which, using (Definition (Interpolating polynomial: characterization)), is implied by:

$$(7)\quad \underset{\text{Poly[K]}}{\in}[\langle a1\rangle] \bigwedge \underset{\text{Poly[K]}}{\deg}[\langle a1\rangle] \le |\langle x1\rangle| - 1 \bigwedge \underset{i=1,\ldots,|\langle x1\rangle|}{\forall} \left(\underset{\text{Poly[K]}}{\text{eval}}[\langle a1\rangle, \langle x1\rangle_i] = \langle a1\rangle_i \right),$$

Formula (7) can now be easily verified by expanding the polynomial operations defined in the functor.

Induction hypothesis: We assume for arbitrary but fixed $n \ge 1$

(8) $\underset{x}{\forall}\,(|x| = n \wedge |a| = |x|) \Rightarrow$ IsInterpolatingPolynomial[NevillePolynomial[x, a, K], x, a, K] ,

and show

(9) $(|x| = n + 1 \wedge |a| = |x|) \Rightarrow$ IsInterpolatingPolynomial[NevillePolynomial[x, a, K], x, a, K] .

We assume

(10) $|x| = n + 1$,

(11) $|a| = |x|$,

From (10) and (11), we can infer:

(12) $x = \langle x0, \overline{x}, xn\rangle$,

(13) $a = \langle a0, \overline{a}, an\rangle$,

for new constants x0, xn, a0, an and new constant sequences $\overline{x}$ and $\overline{a}$ of length $n - 1$.

The prover guesses this particular structure for representing x and a from the definition of NevillePolynomial.

It remains to show

(14) IsInterpolatingPolynomial[NevillePolynomial[⟨x0, $\overline{x}$, xn⟩, ⟨a0, $\overline{a}$, an⟩, K], ⟨x0, $\overline{x}$, xn⟩, ⟨a0, $\overline{a}$, an⟩, K].

Formula (14), using (Algorithm (Neville)), is implied by:

$$(15)\quad \text{IsInterpolatingPolynomial}\Big[\Big(\Big(x \underset{\text{Poly[K]}}{\overline{\text{Poly[K]}}} \underset{\text{Poly[K]}}{\text{const}}[x0]\Big) \underset{\text{Poly[K]}}{*} \text{NevillePolynomial}[\langle\overline{x}, xn\rangle, \langle\overline{a}, an\rangle, K]\Big)_{\overline{\text{Poly[K]}}},$$

$$\Big(x \underset{\text{Poly[K]}}{\overline{\text{Poly[K]}}} \underset{\text{Poly[K]}}{\text{const}}[xn]\Big) \underset{\text{Poly[K]}}{*} \text{NevillePolynomial}[\langle x0, \overline{x}\rangle, \langle a0, \overline{a}\rangle, K]\Big) \Big/_{\text{Poly[K]}} \left(xn \underset{K}{\overline{-}} x0\right),$$

$$\langle x0, \overline{x}, xn\rangle, \langle a0, \overline{a}, an\rangle, K\Big]$$

Membership in the polynomial domain and the degree bound for the interpolating polynomial are not too difficult to prove. For proving the evaluation property we need some auxiliary knowledge about polynomial evaluation, such as e.g. eval[$p + q$, a] = eval[p, a] + eval[q, a], which can be proven by *another special prover*, which can handle formulae containing the $\sum$-quantifier. In our approach of theory exploration, we suppose that this knowledge has already been proven in a previous exploration phase and is for this proof available in the knowledge base KB. After several simplifications we arrive at the following formula to be proved:

$$\underset{i=1,\ldots,n+1}{\forall} \Bigg(\Big(\underset{\text{Poly[K]}}{\text{eval}}\Big[\Big(x \underset{\text{Poly[K]}}{\overline{\text{Poly[K]}}} \underset{\text{Poly[K]}}{\text{const}}[x0]\Big), \langle x0, \overline{x}, xn\rangle_i\Big] \underset{K}{*} \underset{\text{Poly[K]}}{\text{eval}}\Big[$$

$$\text{NevillePolynomial}[\langle\overline{x}, xn\rangle, \langle\overline{a}, an\rangle, K], \langle x0, \overline{x}, xn\rangle_i\Big] \underset{K}{\overline{-}} \underset{\text{Poly[K]}}{\text{eval}}\Big[\Big(x \underset{\text{Poly[K]}}{\overline{\text{Poly[K]}}} \underset{\text{Poly[K]}}{\text{const}}[xn]\Big), \langle x0, \overline{x}, xn\rangle_i\Big] \underset{K}{*}$$

$$\underset{\text{Poly[K]}}{\text{eval}}[\text{NevillePolynomial}[\langle x0, \overline{x}\rangle, \langle a0, \overline{a}\rangle, K], \langle x0, \overline{x}, xn\rangle_i]\Big) \Big/_{K} \left(xn \underset{K}{\overline{-}} x0\right) = \langle a0, \overline{a}, an\rangle_i \Bigg)$$

Since $\langle x0, \overline{x}, xn\rangle_i$ (and $\langle a0, \overline{a}, an\rangle_i$) can be simplified in case $i = 1$ or $i = n + 1$ a case distinction is now made:

Case $i = 1$: We have to show

$$\Big(\underset{\text{Poly[K]}}{\text{eval}}\Big[\Big(x \underset{\text{Poly[K]}}{\overline{\text{Poly[K]}}} \underset{\text{Poly[K]}}{\text{const}}[x0]\Big), x0\Big] \underset{K}{*} \underset{\text{Poly[K]}}{\text{eval}}[\text{NevillePolynomial}[\langle\overline{x}, xn\rangle, \langle\overline{a}, an\rangle, K], x0] \underset{K}{\overline{-}}$$

$$\underset{\text{Poly[K]}}{\text{eval}}\Big[\Big(x \underset{\text{Poly[K]}}{\overline{\text{Poly[K]}}} \underset{\text{Poly[K]}}{\text{const}}[xn]\Big), x0\Big] \underset{K}{*} \underset{\text{Poly[K]}}{\text{eval}}[\text{NevillePolynomial}[\langle x0, \overline{x}\rangle, \langle a0, \overline{a}\rangle, K], x0]\Big) \Big/_{K} \left(xn \underset{K}{\overline{-}} x0\right) = a0$$

which, using (Definition (Polynomial Domain)), is implied by:

(22) $0 \underset{K}{\overline{-}} \left((x0 - xn) \underset{K}{*} a0\right) \Big/_{K} \left(xn \underset{K}{\overline{-}} x0\right) = a0$.

Formula (22) can be verified by auxiliary knowledge on arithmetic in K. The remaining cases proceed analogously.

6 Application of the Algorithm to Concrete Examples

The recursive Algorithm["Neville"] can be used immediately in computations *without any translation* to some machine-executable language. The *Theorema* command "Compute" can perform rewriting using the recursive definition as given in Section 4. Rewriting is done by the interpreter of the underlying Mathematica system. In addition, it can access semantics for the algorithmic language constructs provided by the *Theorema* language (e.g. finite tuples, quantifiers with finite ranges, arithmetic on numbers, etc.), which is needed for performing polynomial arithmetic as defined in the polynomial functor in Section 2.

```
Compute[NevillePolynomial[⟨1, 2, 3, 4, 5⟩, ⟨3, 1, 5, 2, 6⟩, ℚ],
  using → ⟨Definition["Polynomial Domain"], Algorithm["Neville"]⟩]
```

$$\left\langle 51, -\frac{1093}{12}, \frac{443}{8}, -\frac{161}{12}, \frac{9}{8} \right\rangle$$

This computation tells that the Neville-polynomial over $\mathbb{Q}$ for the tuples $\langle 1, 2, 3, 4, 5\rangle$ and $\langle 3, 1, 5, 2, 6\rangle$ is the polynomial $\langle 51, -\frac{1093}{12}, \frac{443}{8}, -\frac{161}{12}, \frac{9}{8}\rangle$, which would commonly be written as the arithmetic term $51 - \frac{1093}{12}\, x + \frac{443}{8}\, x^2 - \frac{161}{12}\, x^3 + \frac{9}{8}\, x^4$.

7 Conclusion

The *Theorema* system has been used in formal development of part of a mathematical theory. An entire exploration cycle—from defining mathematical concepts, over stating mathematical properties, computer-supported proving, until finally applying mathematical knowledge to concrete objects—has been carried through inside the system. The main objective of this case study is to show the integration of *proving* and *computing* in combination with an attractive mathematics-oriented syntax inside *one system*.

References

[AXIOM] Axiom. *Developed by IBM Research, directed by R. Jenks.* http://www.nag.com/symbolic_software.asp.

[Buchberger 96a] B. Buchberger: *Symbolic Computation: Computer Algebra and Logic*. In: Frontiers of Combining Systems (F. Baader, K.U. Schulz eds.), pp. 193-220. Applied Logic Series. Kluwer Academic Publishers, 1996.

[Buchberger 96b] B. Buchberger: *Mathematica as a Rewrite Language*. Invited paper in: Proceedings of the Fuji Conference on Functional Logic Programming, Shonan Village, Nov 1-4, 1996, (T. Ida ed.), pp. 1−13,Telos Publishing.

[MKM 03] Bruno Buchberger, Gaston Gonnet, Michiel Hazewinkel (eds.). *Mathematical Knowledgement Management.* Special issue of the journal Annals of Mathematics and Artificial Intelligence, Kluwer Publishing Company, 2003.

[Windsteiger 99] W. Windsteiger: *Building up Hierarchical Mathematical Domains Using Functors in Theorema*. In: A. Armando and T. Jebelean, editors, Electronic Notes in Theoretical Computer Science, vol 23-3, p. 83-102, Elsevier, 1999.

Using *Theorema* in the Formalization of Theoretical Economics*

Manfred Kerber[1], Colin Rowat[2], and Wolfgang Windsteiger[3]

[1] Computer Science
[2] Economics
University of Birmingham, Birmingham B15 2TT, England
[3] RISC
JKU Linz, 4232 Hagenberg, Austria

Abstract. Theoretical economics makes use of strict mathematical methods. For instance, games as introduced by von Neumann and Morgenstern allow for formal mathematical proofs for certain axiomatized economical situations. Such proofs can—at least in principle—also be carried through in formal systems such as *Theorema*. In this paper we describe experiments carried through using the *Theorema* system to prove theorems about a particular form of games called pillage games. Each pillage game formalizes a particular understanding of power. Analysis then attempts to derive the properties of solution sets (in particular, the core and stable set), asking about existence, uniqueness and characterization.

Concretely we use *Theorema* to show properties previously proved on paper by two of the co-authors for pillage games with three agents. Of particular interest is some pseudo-code which summarizes the results previously shown. Since the computation involves infinite sets the pseudo-code is in several ways non-computational. However, in the presence of appropriate lemmas, the pseudo-code has sufficient computational content that *Theorema* can compute stable sets (which are always finite). We have concretely demonstrated this for three different important power functions.

1 Introduction

Theoretical economics may be regarded as a branch of applied mathematics, drawing on a wide range of mathematics to explore and prove properties of stylized economic environments. Since the Second World War, one particularly important body of theory has been game theory, as introduced by von Neumann and Morgenstern [17] and successfully developed by John Nash.

* The first author would like to thank the 'Bridging The Gap' initiative under EPSRC grant EP/F033087/1 for supporting this work. The first and second author would like to thank the JKU Linz for its hospitality.

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 58–73, 2011.

The game theory stemming from von Neumann and Morgenstern has become known as cooperative game theory; it allows abstraction from the details of how agents might interact, instead focusing directly on how final outcomes may or may not dominate each other. As dominance is a binary relation, cooperative game theory has lent itself naturally to axiomatic analyses. The game theory stemming from Nash has become known as non-cooperative game theory; it is explicitly constructive, requiring specification of a game form that details the set of permissible moves available to agents. Solutions to cooperative games have been difficult to calculate relative to non-cooperative games, contributing to the latter body of theory's current preeminence within game theory.[1]

In the current work, we use *Theorema* [19] to formalize a particular cooperative game form, called pillage games, and to prove formally certain properties of them. Pillage games, introduced in [8], form an uncountable set of cooperative games, taking the two best known classes of cooperative games (those in characteristic and partition function form) as boundary points. At the same time, even though each is defined over an uncountable domain, their structure has thus far been sufficient to avoid Deng and Papadimitriou's [5] pessimistic conclusions that whether a solution even exists may be undecidable. Thus, pillage games provide a class of games for analysis that is simultaneously rich and tractable. To our knowledge, this represents the first attempt to formally prove properties of a cooperative game. Related is work on other axiomatic proofs within economic theory which have been formalized. Arrow's theorem in social choice has attracted the most attention, including studies by Wiedijk [18] using *Mizar*, Nipkow [15] using *HOL*, and Grandi and Endriss [6] using *Prover9*. Non-cooperative game theory has also received attention, including by Vestergaard and co-authors [16] with *Coq*.

The formalization of a particular game form in economics can be interesting to computer science for at least two reasons, and to economics for at least one. For computer science, economics is a relatively new area for automated theorem proving and therefore presents a new set of canonical examples and problems. Secondly, it is an area which typically involves new mathematics in the sense that axioms particular to economics are postulated. Further, in the case of pillage games, the concepts involved are of a level that an undergraduate mathematics student can understand easily. That is, the mathematics is of a level that should be much more amenable for formalizations than research level mathematics.

For economics, as in any other mathematical discipline [11], establishing new results is typically an error-prone process, even for the most respected researchers. We cite but two examples from cooperative game theory:

1. In founding game theory, von Neumann and Morgenstern [17] assumed that one of the key concepts of the field, the so-called stable set (by them just called the "solution"), always existed in games in characteristic function form. This was subsequently demonstrated by Lucas [12] to be incorrect.

[1] For example, Gambit [14] solves a broad class of non-cooperative games.

2. Nobel Prize winning economist and game theorist Maskin [13] claimed that certain properties of a game in partition function form extended from $n = 3$ to $n > 3$. However, de Clippel and Serrano [4] found counterexamples with $n > 3$.

It is understandable that such problems occur since typically for any new axiom set humans have initially no or only limited intuition. This way it is easy to assume false theorems and to overlook cases in proofs. Proofs found in mathematics in general and theoretical economics in particular, can be viewed from a logical point of view more like proof plans. That is, not all details are given, hidden assumptions may be overlooked, proof steps may be incorrect, generalizations may not hold. Thus, any mathematical discipline, including theoretical economics, can benefit from formalizing proofs since this will make proofs much more reliable. However, there are other potential benefits. For instance, in experimenting with axiomatizations it is much easier to reuse proof efforts. Furthermore the dependencies of assertions can be accessed more easily and experiments with the computational content of theorems becomes possible which without computer support would be time consuming and error-prone.

In this work we report on our experiments with the *Theorema* system which we conducted to formalize pillage games, to prove certain properties of them, and to exploit computational features in them. Full verification of the formal statements is an important goal of these experiments. However, we also look at less labour extensive ways of making use of *Theorema* and will discuss this.

The paper has the following structure. In the next section we give a brief introduction to pillage games. In Section 3 we present the representations in *Theorema* and the proofs of some formal statements which we formally proved in *Theorema*. A focus of the presentation as presented in Subsection 3.4 is the pseudo-code, which summarizes the results of the paper we formalize. This pseudo-code is non-computational in several ways. However, a mixture of proof and computation makes it possible to evaluate it in concrete cases. In Section 4 we evaluate the approach taken.

2 A Brief Introduction to Pillage Games

The class of games used for our experiments are called pillage games, a particular form of cooperative games, introduced by Jordan in [8]. In a nutshell, pillage games describe how *agents* (n in total) can form coalitions in order to redistribute all or part of the possessions of other coalitions among themselves. The possessions are described by a so-called *allocation*, a vector of n non-negative numbers which sum to one. Given two such vectors, x and y, three coalitions are induced: the *win set* of a transition from x to y is $\{i \mid y_i > x_i\}$; the *lose set* is $\{i \mid x_i > y_i\}$; the remaining agents are indifferent between x and y. Pillage is possible if and only if the win set is more powerful at x than is the lose set; if this is so, it is said that y *dominates* x.

The power of a coalition is determined by a so-called *power function*, π, a function which depends exclusively on the coalition members and the holdings of all agents. A power function must satisfy three monotonicity axioms: firstly *weak coalition monotonicity (WC)*, whereby taking a new member into a coalition does not decrease the coalition's power; secondly *weak resource monotonicity (WR)*, whereby weakly increasing the holdings of a coalition's members does not decrease the coalition's power; and thirdly, *strong resource monotonicity (SR)*, whereby strictly increasing the holdings of a coalition's members strictly increases the coalition's power. This setting given, two sets are of interest. Firstly, the *core*, the set of undominated allocations, and secondly the *stable set*, a set of allocations such that none dominates another (called *internal stability*) but at least one dominates each non-member allocation (called *external stability*). The core always exists, and is unique, but may be empty. A stable set may not exist since it has to satisfy two conflicting properties, on the one hand it must contain sufficiently many elements so that any element not in it is dominated by an element in it (e.g. the empty set is not externally stable), on the other hand it must not contain so many elements that none dominates another (e.g. the full set of all allocations is not internally stable in a pillage game). If a stable set exists, it is finite and contains the core; uniqueness has been established for some pillage games, and no counterexamples have been found as yet. If agents are forward looking, expecting that y dominating x may allow a subsequent comparison between z and y. Jordan [8] proved that a stable set is a *core in expectation*, a set of undominated allocations given some such future expectation (and consequent comparison of x and z, rather than x and y).

Jordan [8] proved general properties about pillage games (such as the finiteness of the stable set) and investigated the possibilities of three particular power functions for arbitrarily many agents. Kerber and Rowat [10] studied an infinite class of power functions for three agents. In particular they gave a complete characterization of the stable set for arbitrary power functions (with three agents) which satisfy three additional axioms. The first axiom is *continuity* in the resources; although definable with an ϵ-δ-statement, the intermediate value theorem is actually used. The second axiom, *responsiveness*, requires that a coalition gaining a member with some power as a singleton strictly increases the coalition's power. The third axiom, *anonymity*, requires that power, dominance, and—consequently—the stable set are invariant under permutations of the agents; thus, an agent's identity is irrelevant to a coalition's power, merely its presence or absence, and its holdings are relevant.

3 Formalizations in *Theorema*

Within this case study, we fully proved the first three lemmas from [10]. Furthermore, we give a formalization of the pseudo-code that summarizes the results derived in that paper. In this section, we briefly give the main assertions,

which we proved in *Theorema* input syntax.[2] We begin with the definition of a power function. In all what follows, $I[n] := \{1, \ldots, n\}$ stands for the set of n agents and $X[n]$ denotes the set of all allocations for n agents.

With these preliminaries we use *Theorema* to formally define *weak coalition monotonicity (WC)*, *weak resource monotonicity (WR)*, *strong resource monotonicity (SR)*, and *power function* as follows.[3]

Definition. ["WC", any$[\pi, n]$, bound[allocation$_n[x]$],

$$\mathrm{WC}[\pi, n] :\Leftrightarrow n \in \mathbb{N} \wedge \forall_{\substack{C1,C2 \\ C1 \subset C2 \wedge C2 \subseteq I[n]}} \forall_{x} \pi[C2, x] \geq \pi[C1, x]]$$

Definition. ["WR", any$[\pi, n]$, bound[allocation$_n[x]$, allocation$_n[y]$],

$$\mathrm{WR}[\pi, n] :\Leftrightarrow n \in \mathbb{N} \wedge (\forall_{\substack{C \\ C \subseteq I[n]}} \forall_{x,y} ((\forall_{i \in C} y_i \geq x_i) \implies \pi[C, y] \geq \pi[C, x]))]$$

Definition. ["SR", any$[\pi, n]$, bound[allocation$_n[x]$, allocation$_n[y]$],

$$\mathrm{SR}[\pi, n] :\Leftrightarrow n \in \mathbb{N} \wedge (\forall_{\substack{C \\ C \subseteq I[n] \wedge C \neq \emptyset}} \forall_{x,y} ((\forall_{i \in C} y_i > x_i) \implies \pi[C, y] > \pi[C, x]))]$$

Definition. ["powerfunction", any$[\pi, n]$, powerfunction$[\pi, n] :\Leftrightarrow \wedge \left\{ \begin{array}{l} \mathrm{WC}[\pi, n] \\ \mathrm{WR}[\pi, n] \\ \mathrm{SR}[\pi, n] \end{array} \right.$]

In this formalization, we decided not to put explicit conditions on variables in definitions assuming that defined expressions will only be used "as intended

[2] The *Theorema* language syntax comes very close to how mathematicians are used to write up things. In particular, two-dimensional notation can be used both in input and output through *Mathematica*'s notebook technology, so that a *Theorema* formalization can easily be read and understood by a mathematician. In this presentation, we typeset all *Theorema* expressions in LaTeX with the aim to mimick their appearance in *Theorema* as closely as possible. Formal text blocks (definitions, theorems, lemmas, etc.) in *Theorema*, so-called environments, are of the form '*Env*[l, any[v], with[C], bound[r], *form*]', where *Env* is the type of environment, l is a string label used to refer to this environment, v lists the universal variables in *form*, C is a formula expressing a condition on v, r specifies ranges for bound variables in *form*, and finally *form* is a single formula or a sequence of formulae. After evaluating a formal text block in a *Theorema* session, it can be referred to (e.g. in a call to a prover) by '*Env*[l]'. Concretely, e.g. in (WC), the 'any$[\pi, n]$' makes the definition applicable for all π and all n and the 'bound[allocation$_n[x]$]' makes the 'for all x' to actually range over all allocations x of length n. The real convenience of the 'bound'-construct will be revealed once we collect several definitions into one 'Definition'-environment, e.g. one definition for the axioms (WC), (WR), and (SR), where one 'bound'-statement would suffice for all three axioms. For more details we refer to [1].

[3] For full formalizations in *Theorema* together with proofs see also `http://www.cs.bham.ac.uk/~mmk/economics/theorema`.

by definition." In theorems and lemmas, of course, we explicitly list all preconditions. Furthermore, we split the definitions into individual environments, although the *Theorema* language would allow to collect several formulae into one environment. We decided to proceed this way because we later want to use the definitions on an individual basis, and the current version of *Theorema* allows to access only whole environments, not single formulae.

3.1 Lemma 1: Representation

The first lemma states that the power of a coalition depends only on the holdings of the members of the coalition, but not on the holdings of the other agents.

Lemma["powerfunction-independent", any$[\pi, n, C, x, y]$,
with[allocation$_n[x] \wedge$ allocation$_n[y] \wedge C \subseteq I[n] \wedge$ powerfunction$[\pi, n]$],
$\underset{i \in C}{\forall} (x_i = y_i) \implies (\pi[C, x] = \pi[C, y])$]

In order to prove this, we call the *Theorema* predicate logic prover, see [2], by

Prove[Lemma["powerfunction-independent"],
using→{Definition["powerfunction"],Definition["WR"], Proposition["ref/as"]},
by → PredicateProver, SearchDepth → 100],

which uses the definition of power function, the axiom (WR), and the following (trivial) property of the partial ordering $\geq$ on real numbers in its knowledge base:[4]

Proposition. ["ref/as", any$[a, b]$, $(a = b) \Leftrightarrow (a \geq b \wedge b \geq a)$].

With these settings the lemma is proved fully automatically. It also generates a proof if it is given not only the axioms it needs for a proof but the full theory. *Theorema* generates a human readable proof of the proof search, which is ten pages long for a proof with the full theory, and five pages for the setting used above. This proof can be automatically tidied to a three page proof, which contains only those steps necessary for the final argument, see Fig. 1 to get an impression of how *Theorema* presents a human readable proof (further information is at `http://www.cs.bham.ac.uk/~mmk/economics/theorema`).

3.2 Lemma 2: Domination

The second Lemma states that when the opposing coalitions consist of one element each and the power function is anonymous then the coalition which wins is the one with bigger holdings. In order to formalize Lemma 2 we need in addition to formalize the win set of a transition from an allocation x to an allocation y (those agents which benefit from the transition) and the lose set (those agents to whose detriment the transition is) as well as the notions of domination (x domintates y if the coalition consisting of the win set is, in x, more powerful

[4] This proposition can in turn be proven fully automatically using reflexivity of $\geq$ for the part from left to right and anti-symmetry of $\geq$ for the opposite direction.

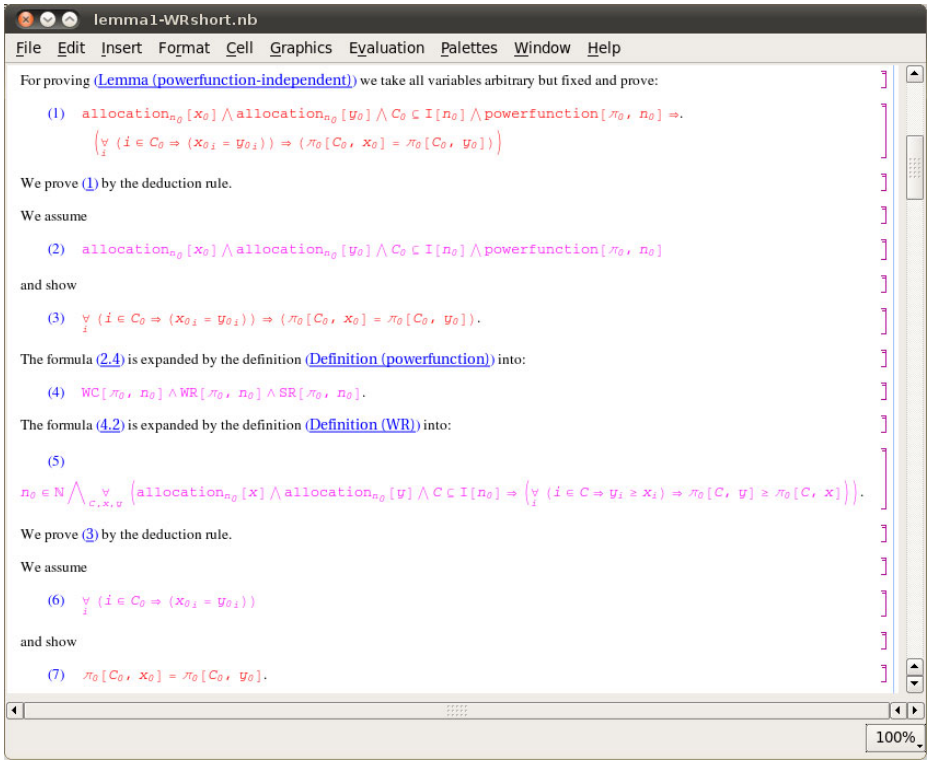


Fig. 1. A human readable proof in *Theorema*

than the coalition consisting of the lose set) and anonymity (invariance under permutations of agents). See also Section 2.

Definition. ["WinLose", any$[n, x, y]$,
$W[n, x, y] := \{i \mid_{i \in I[n]} y_i > x_i\}$ "W"
$L[n, x, y] := \{i \mid_{i \in I[n]} x_i > y_i\}$ "L"]

Definition. ["domination", any$[\pi, n, x, y]$,
dominates$[y, x, \pi, n] :\Leftrightarrow n \in \mathbb{N} \wedge \pi[W[n, x, y], x] > \pi[L[n, x, y], x]$]

Definition. ["anonymity", any$[\pi, n]$, bound[allocation$_n[x]$, allocation$_n[y]$]],

$$\text{anonymous}[\pi, n] :\Leftrightarrow n \in \mathbb{N} \wedge \underset{\substack{\sigma \\ \text{permutation}[\sigma, I[n]]}}{\forall} \underset{\substack{Cx, Cy, x, y \\ Cx \subseteq I[n] \wedge Cy \subseteq I[n]}}{\forall} \underset{i}{\forall} (((i \in Cx) \iff (\sigma[i] \in Cy)) \wedge (x_i = y_{\sigma[i]})) \implies (\pi[Cx, x] = \pi[Cy, y])]$$

Lemma 2 can then be formulated as follows:

Lemma. ["ANdominates", any$[n, x, y]$, with$[n \in \mathbb{N} \wedge n \geq 2 \wedge \text{allocation}_n[x] \wedge \text{allocation}_n[y] \wedge (W[n, x, y] = \{1\}) \wedge (L[n, x, y] = \{2\})]$,
$$\underset{\substack{\pi \\ \text{anonymous}[\pi,n] \wedge \text{powerfunction}[\pi,n]}}{\forall} (\text{dominates}[y, x, \pi, n] \Leftrightarrow x_1 > x_2)\,]$$

While this is apparently intuitively correct, a formal argument is not completely trivial. Since anonymity involves permutations of the agents, a formal proof requires auxiliary knowledge about permutations. In the concrete case, we provide a lemma about permuted tuples, namely that when swapping the first two elements in a tuple the second element in the new tuple is equal to the first of the original:

Lemma. ["perm swap", any$[x]$, $\text{perm}[x, \sigma_{1,2}]_2 = x_1$],

where $\text{perm}[x, \sigma]$ stands for the tuple x permuted by σ, and $\sigma_{1,2}$ is the permutation swapping the first and second component while leaving the rest unchanged. Furthermore, we use a lemma saying that the power does not change when swapping agents 1 and 2, i.e.

Lemma. ["ANswap", any$[n, \pi, x]$, with$[\text{anonymous}[\pi, n] \wedge \text{allocation}_n[x]]$,
$\pi[\{1\}, x] = \pi[\{2\}, \text{perm}[x, \sigma_{1,2}]]$].

Lemma["perm swap"] can be verified without effort based on the definitions only, whereas Lemma["ANswap"] needs the definitions of the concepts involved plus idempotency of swapping, which is trivial but still automatically verified based on the definition of $\sigma_{1,2}$. These proofs, and all that will follow, have been generated by the *Theorema* set theory prover, see [20]. Equipped with Lemma["perm swap"] and Lemma["ANswap"] in the knowledge base, a three page human-understandable proof of the two directions of Lemma 2 is obtained. Note, however that the introduction of the auxiliary lemma is a eureka step, since finding the right permutation is key to the proof of our main lemma; it is difficult to see how *Theorema*—with its currently integrated theorem provers—could automatically find this permutation. This shows that the main idea of the proof requires a fairly intuitive understanding of permutations which needed to be made more explicit when instructing *Theorema*.

The crucial question is, of course, how to obtain appropriate intermediate lemmas. One approach in this kind of *theory exploration* is to always prove all properties of interactions between available concepts before introducing a new concept. This approach is advocated for instance in [3]. In the concrete case, it would require proving many properties of permuted vectors and swaps before talking about anonymity. *Theorema*, however, also supports a much more goal-oriented approach: in the case of a failing proof attempt, it displays the partial proof, allowing a human check of where resources are used, and whether

Theorema can be better guided. In most cases, formulating an appropriate lemma is then a straightforward exercise, so that even this can be automated. There is a literature on lemma speculation from failing proofs (see e.g. [7]); some mechanisms have been implemented in the *Theorema* system as well (see [1]). In this case study, these tools have not been employed.

3.3 Lemma 3: The Core

The *dominion* $D[Y, \pi, n]$ denotes the set of all allocations that are dominated by an allocation in Y. Using this, the *core*, i.e. the set of undominated allocations, can simply be defined as $K[\pi, n] := X[n] \setminus D[X[n], \pi, n]$. The third lemma specializes to the case of three agents and consists of two parts. Firstly, if the core is empty then the tyrannical elements (one agent possesses everything) are dominated by the half splits (e.g., $\langle 1/2, 1/2, 0\rangle$ dominates $\langle 0, 0, 1\rangle$):

Lemma. ["Core,n=3,a", any[π],
$(K[\pi, 3] = \emptyset) \implies \underset{i,j,k \in I[3]}{\forall} (\text{distinct}[i, j, k] \implies t[i, 3] \in D[\{s[j, k, 3]\}, \pi, 3])$].

On the other hand, for anonymous power functions, the half way splits are never dominated by the tyrannical elements (e.g., $\langle 0, 0, 1\rangle$ does not dominate $\langle 1/2, 1/2, 0\rangle$), written in *Theorema* as follows:

Lemma. ["Core,n=3,b", any[π], with[anonymous[$\pi, 3$] $\wedge$ powerfunction[$\pi, 3$]],
$\underset{i,j,k \in I[3]}{\forall} (\text{distinct}[i, j, k] \implies s[j, k, 3] \notin D[\{t[i, 3]\}, \pi, 3])$].

It is clear by the definitions of D and domination, that the win and lose sets under tyrannical elements and the half splits, e.g. $W[3, t[i, 3], s[j, k, 3]]$, will play an essential role in the proofs. We then use the computational capabilities of *Theorema* in order to get some intuition about these entities. Using the built-in computational semantics of the *Theorema* language, one can do some experiments like computing $W[3, t[1, 3], s[2, 3, 3]]$ and $L[3, t[1, 3], s[2, 3, 3]]$ resulting in $\{2, 3\}$ and $\{1\}$, respectively. After some calculations of this kind, the following generalization can be conjectured as a lemma:

Lemma. ["s dominates t", any[$i, j, k \in I[3]$], with[distinct[i, j, k]],
$W[3, t[i, 3], s[j, k, 3]] = I[3] \setminus \{i\}$
$L[3, t[i, 3], s[j, k, 3]] = \{i\}$],

The proof of this Lemma can be done by pure computation, since the universal quantifier over i, j, k boils down to just testing finitely many cases, namely the six distinct choices of $i, j, k \in I[3]$. The neat integration of proving and computing in the *Theorema* system is of great value in this kind of investigation, because the statements need no reformulation when switching between proving and computing.

For proving the first part of Lemma 3, instead of going back to the definition of the core, we use a theorem of Jordan that connects the core and the power

of tyrannical allocations, see [8]. Together with Lemma["s dominates t"], this proof goes through without further complications. For the second part, *Theorema* comes up with an indirect proof using a variant of Lemma["s dominates t"] with the roles of s and t interchanged and specialized versions of the axioms (WC) and (SR) and of Lemma["ANswap"] (introduced in the context of Lemma 2), for the case $n = 3$.

3.4 Pseudo-Code and Its Computational Content

The main result of [10] is a classification of the possibilities which a stable set can have in three agent pillage games with continuous, responsive, and anonymous power functions. No stable set may exist but—if one does—it may have up to 15 elements. How these elements are determined (in dependency of π) can be summarized in form of some pseudo-code. This pseudo-code can be represented as an algorithm in *Theorema* as shown in Fig. 2.

Algorithm["StableSet2", any[π],

$$
\begin{array}{l}
\text{stableSet}[\pi] := \\
\left\{\begin{array}{ll}
\left\{\begin{array}{ll}
\text{“no stable”} & \Leftarrow \text{empty}[R[1,\pi]] \\
\text{where}[\mathcal{S} = \text{dyadicSet}[0,3] \cup \bigcup_{i=1,\ldots,3} S[i,\pi], & \\
\quad \left\{\begin{array}{lll}
\mathcal{S} \cup P[\pi] & \Leftarrow \neg\text{fullSet}[\mathcal{S} \cup D[\mathcal{S},\pi,3]] & \Leftarrow \neg\text{empty}[R[1,\pi]] \Leftarrow (*) \quad] \\
\mathcal{S} & \Leftarrow \text{fullSet}[\mathcal{S} \cup D[\mathcal{S},\pi,3]] \quad] & \\
\text{“unknown X”} & \Leftarrow \text{otherwise} &
\end{array}\right. & \\
\text{“unknown R”} & \Leftarrow \text{otherwise}
\end{array}\right. \\
\text{dyadicSet}[1,3] \backslash \text{dyadicSet}[0,3] \qquad \Leftarrow \text{otherwise}
\end{array}\right.
\end{array}
$$

with $(*)$ to be replaced by $\pi[\{1\}, t[1,3]] \geq \pi[\{2,3\}, t[1,3]]$.

Fig. 2. Algorithm to compute a stable set written in *Theorema* notation

The algorithm in Fig. 2 makes use of several sets and conditions which we explain only partly in the following, since not all details are of importance in this context. Important is that certain sets (such as dyadicSet$[1,3]$) are computational, whereas others (such as $R[1,\pi]$) are not. For details of these constructs see [10].

This algorithm is non computational in several ways. For $n = 3$, however, it is a significant improvement on the Roth-Jordan [9] algorithm to determine stable sets, since the latter is non-computational in even more ways.[5] In this section we will discuss how the algorithm, which is written in the first place to summarize

[5] First, the Roth-Jordan algorithm starts with the core without giving an effective means for computing it. Second, for an empty core it provides no clue for finding an initial iterate. Third, the iteration step is not computational since it involves the computation of undominated sets, which are typically infinite. Finally, it is not clear whether terminating at a set $\mathcal{S}$ which is not externally stable means that no stable set exists, or merely that further steps must be taken independently of the algorithm. For details, see [9] or [10].

the results in a concise form, can be used to determine stable sets by a mixture of proving and computing.

We have tested the implementation for the three specific power functions introduced by Jordan [8], *strength in numbers* (SIN), *Cobb-Douglas* (CD), and *wealth is power* (WIP).

If a concrete power function π is given, the algorithm has first to test condition $(*)$. Since we assume that π is computable, both $\pi[\{1\}, t[1,3]] \in [0,1]$ and $\pi[\{2,3\}, t[1,3]] \in [0,1]$, hence $\pi[\{1\}, t[1,3]] \geq \pi[\{2,3\}, t[1,3]]$ can be decided. If the condition is false the stable set is given by the last line of the algorithm.[6] It is computed by *Theorema* as the set $\{\langle 1/2, 1/2, 0\rangle, \langle 1/2, 0, 1/2\rangle, \langle 0, 1/2, 1/2\rangle\}$. This case was concretely tested for the 'strength in numbers' power function, defined as

$$\mathrm{SIN}\pi_\nu[C,x] := \sum_{i\in C} (x_i + \nu)$$

(with $\nu > 1$) for the concrete value of $\nu = 2$.

If the condition is true, however, the algorithm must check whether a particular set $R[1,\pi]$ is empty or not. For finite $R[1,\pi]$, the ad-hoc method to test $R[1,\pi] = \emptyset$ is to compute $R[1,\pi]$ (by enumerating its elements) and then comparing it to the empty set $\emptyset$, which can all be done in a *Theorema* computation. Unfortunately, the set $R[1,\pi]$ is defined in terms of the set $M[1,\pi]$, which in turn is defined in form of the set $B[1,\pi]$ of all triples, where agent 1 on its own is equally powerful as agents 2 and 3 together, i.e.

$$B[1,\pi] := \{x \in X[3] \mid \pi[\{1\}, x] = \pi[I[3]\backslash\{1\}, x]\}\,.$$

The set $M[1,\pi]$ is then a subset of $B[1,\pi]$ such that the x_1 are maximal, and $R[1,\pi]$ those elements in $M[1,\pi]$ which are maximal from the viewpoint of agents 2 and 3. Since $B[1,\pi]$ is typically infinite, testing $R[1,\pi] = \emptyset$ by computation as described above would fail. Without any further knowledge on $R[1,\pi]$, the algorithm returns "unknown R", which indicates that it has insufficient knowledge on the set R and for this reason insufficient knowledge to determine the stable set.

If, however, for the concrete power function π there is additional knowledge that allows us to decide $R[1,\pi] = \emptyset$ (without actually computing $R[1,\pi]$), the algorithm may make use of this knowledge and continue. Concretely in the algorithm above, we provide the following lemma:
Lemma["emptyR,Cobb Douglas", any[ν], empty[$R[1, \mathrm{CD}\pi_\nu]$]],
where $\mathrm{CD}\pi_\nu$ is the 'Cobb-Douglas' power function defined as

$$\mathrm{CD}\pi_\nu[C,x] := |C|^\nu \Big(\sum_{i\in C} x_i\Big)^{1-\nu} \quad \text{for } 0 \leq \nu \leq 1.$$

[6] The allocations in which each agent has either nothing or $\left(\frac{1}{2}\right)^j$ for some integer $j \leq i$ form a so-called *dyadic set*, represented as dyadicSet$[i,n]$ (for n agents). For three agents the previously mentioned tyrannic allocations and half-splits as well as the ones of type $\langle 1/2, 1/4, 1/4\rangle$ play an important role in determining the stable set.

In this case, according to the algorithm, no stable set exists. Note, however, that the knowledge must be formulated appropriately. For instance, in the algorithm above it was not possible to use the usual *Theorema* notation $R[1,\pi] \neq \emptyset$ instead of $\neg\text{empty}[R[1,\pi]]$: if $R[1,\pi] \neq \emptyset$ was given as an auxiliary lemma, the computation engine would have had to process negated equalities in an appropriate manner, which the current version of *Theorema* is not capable of.

If $R[1,\pi]$ is known not to be empty, further knowledge is necessary. Most importantly, does the current iteration of the computed candidate stable set together, with the allocations dominated by it, include all possible allocations? Formally, is $\text{fullSet}[\mathcal{S} \cup D[\mathcal{S},\pi,3]]$ true or false? This will typically not be computational, since $D[\mathcal{S},\pi,3]$ is infinite. Hence for any particular given power function π a corresponding lemma is necessary, which states whether the property holds or not. In case of the 'wealth is power' power function, defined as

$$\text{WIP}\pi[C,x] := \sum_{i \in C} x_i,$$

the property does not hold since there are three points which are not dominated by the allocations computed so far. In this case, the stable set is computed by *Theorema* to $\mathcal{S} \cup P[\pi]$, which is evaluated directly, since $P[\pi]$ is a set of at most three elements explicitly defined in [10].

The algorithm presented above does not check whether a power function satisfies the additional axioms (continuity, responsiveness, and anonymity), or even whether the functions supplied are actually power functions (satisfying axioms WC, WR, and SR). As a consequence, the algorithm may give wrong answers if applied inappropriately. This can be remedied by adding further conditions to the functions. While this makes the application safer on the one hand, it increases the proof obligations on the other hand.

Note that when applying the algorithm to concrete examples, part of the knowledge may typically be computed, certain information about sets derived from $R[1,\pi]$ must be given in form of lemmas. That is, the algorithm consists of a mixture of proving and computing. It is conceivable that *Theorema* could be extended to make use of the underlying *Mathematica* system to compute the sets $B[i,\pi]$ for particular power functions π.

4 Added Value—Price Paid

In this section we summarize the added value from the point of view of a researcher in theoretical economics. The added value is partly due to the formalization effort and could have been achieved with any formal system, partly, however, it is specifically due to *Theorema* and its features. Furthermore we discuss the effort necessary to do such a formalization.

An obvious point to mention is the greater precision that a formal system requires. This is an advantage (enhanced clarity, greater reliability) and at times a disadvantage (greater effort) at the same time. A concrete example where the formal precision played a role was the characterization of the core as

$K = \{x \in X \,|x_i > 0 \Leftrightarrow \pi(\{i\}, x) \geq \pi(I \backslash \{i\}, x)\}$. In this definition a free index i is used. Two standard interpretations are possible, an existentially quantified one ('there is an i such that'), or a universally quantified one ('for all i holds'). When translating into *Theorema* first the existentially quantified translation was chosen. However, this was later found to be the wrong one when it was used.

Another obvious advantage is that we can have much higher confidence in lemmas and theorems which are formally proved. In addition, the system clearly states all the knowledge used for proving a particular assertion and any hidden assumptions have to be made explicit. On the other hand, an automated theorem prover will typically be inundated with too much information so that the knowledge used to prove an assertion is typically minimal. This is useful knowledge since it allows us to generalize statements.

There are also particular advantages of the computational aspects of *Theorema*. They allow computation and checking particular structures for specific examples. For instance, for a given power function π it is possible to compute particular sets (such as $R[i, \pi]$ or $P[\pi]$) and to see whether these correspond to the intuition. If they do, this gives confidence that the formalization accurately mirrors the intuition. If they do not, then either the intuition needs to be changed or the formalization does not reflect what actually should have been formalized and needs to be changed. Of course, also incorrect statements may be discovered this way. For instance, it led to an adjustment of the algorithm in Fig. 2 in which the last case was incorrectly copied from the corresponding lemma into it. That is, a mistake in the algorithm was detected, although no attempt was made to verify it.[7]

A particular advantage of using *Theorema* is due to the fact that some reasoners—in particular the set theory prover used mostly for our study—use an interface to the computation engine, so that proving and computing are well-integrated in the *Theorema* system. In this case study, this feature turned out to be useful when the whole proof of Lemma["s dominates t"] was shifted to just one simple computation on finite sets. In our concrete application example, we also make use of the computational parts of the algorithm in Fig. 2 to determine the stable set. The algorithm contains algorithmic parts but needs at two steps an oracle which can be given in form of lemmas. These can in turn be formally proved in *Theorema*. The possibility of *Theorema* working in a 'compute' mode makes it relatively painless to combine reasoning and computation. This makes it possible to determine the stable set for concrete power functions by a mixture of reasoning and computation.

As mentioned in the previous section it seems feasible to allow *Theorema* to move more tasks from its reasoning part into its computation part (for specialized power functions). One way to achieve this is to represent infinite sets finitely. More work in this direction is necessary.

Obviously using a system such as *Theorema* has a price. The formalization is more labour intense than formalizing the knowledge just on paper or using

[7] As the algorithm summarizes the central results in [10], we are still in the process of verifying the proofs in paper—informally and formally.

LaTeX. However, just writing down the definitions, lemmas, and theorems in *Theorema* is—after an introduction phase of a day—almost as painless as using LaTeX. Formalizing the knowledge is relatively easy, formally proving the lemmas and theorems, however, is typically labour intense and requires knowledge which cannot be acquired so quickly. Certain formalizations need to be changed in order to avoid certain pitfalls of the reasoners. Additionally it is necessary to know which integrated prover to use best and to adjust it by setting suitable options. This requires expert knowledge. Furthermore, it may be necessary to introduce suitable auxiliary lemmas to prove statements. This makes it currently unlikely that there will be a big uptake in using such systems, although in balance the extra work required—at least for the formal properties proved in this work—does not look too huge with one to two days of work for a formal statement. This effort may go up as the statements get more complicated as the paper proceeds, which is partly due to the fact that typically authors acquire some intuition about the concepts introduced in earlier sections, and this intuition is rarely made explicit by additional lemmas or corollaries. That is, this increased effort could be considered as a price paid by the author and an added value on the side of the reader of a formalized paper. It should be noted that just using *Theorema* for representing knowledge and making use of this knowledge, e.g. by evaluating the algorithm for concrete power functions can uncover mistakes and thereby improve the reliability of the results.

5 Conclusion

The true benefit of a formalization can only be obtained if it is used. Typically you either prove something with it, or you use it in some computation. The latter is only possible in finite domains. In the examples, for instance, it meant that once a set is reduced to a finite set of concrete values, lemmas need not be formally proved as their truth status can be shown by exhaustive computation of all possible cases. This dual feature of computation and proving also allowed us to use the algorithm for computing the stable set for power functions for which certain features had been established (or claimed) by lemmas. Note that *Theorema* does not insist on proving assertions claimed. This makes it very easy to cite external theorems (concretely, in this example, theorems established by Jordan previously) without reproving them. While this is very convenient, it has a downside, namely it is easy to base a theorem on a lemma which is not proved or is even wrong.

A very useful feature of *Theorema* is the possibility to generate incomplete proofs. This allows both detection of potential problems with the formalizations (e.g. inconsistent argument orderings) and monitoring of whether the integrated *Theorema* prover used is making useful steps towards a solution. Moreover, the study of an incomplete proof typically helps greatly in the formulation of lemmas that then help the prover to succeed in a subsequent run, which is quite helpful since the integrated provers of *Theorema* are typically not interactive. This means, that, if they do not find a proof fully automatically, then the only

possibility to guide the search is to adjust suitable options (which requires good knowledge of the inner workings of such a prover), or to introduce additional auxiliary lemmas.

As with any other theorem proving system, applying *Theorema* requires good knowledge of the system. On the positive side, the *Theorema* input language is very flexible and allows—after a very brief introduction to the system—for a natural representation which is close to a paper representation. Care should, however, be taken, since writing down statements in *Theorema* does not mean that they are correct, or can be directly used in the form they have been inputted. The input language is untyped which makes it easy to write things down. However, it also means that it is easy to introduce mistakes, e.g., *Theorema* would not complain if you defined a power function with the argument order as in powerfunction$[\pi, n]$ and later used it in form of powerfunction$[n, \pi]$. However, proofs may not be established any more in such a case.

While proving assertions formally using the *Theorema* system requires typically good knowledge of the proof in the first place, the formalization effort pays off in several ways. Above all it is possible to gain greater confidence in the correctness of the assertions, and it is possible to make experiments—concretely with specific instances of power functions—which otherwise are labour intense and error-prone.

References

1. Buchberger, B., Dupre, C., Jebelean, T., Kriftner, F., Nakagawa, K., Vasaru, D., Windsteiger, W.: The Theorema Project: A Progress Report. In: Kerber, M., Kohlhase, M. (eds.) Proceedings of CALCULEMUS 2000, Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, pp. 98–113. A.K. Peters, Natick (2000)
2. Buchberger, B., Craciun, A., Jebelean, T., Kovacs, L., Kutsia, T., Nakagawa, K., Piroi, F., Popov, N., Robu, J., Rosenkranz, M., Windsteiger, W.: Theorema: Towards Computer-Aided Mathematical Theory Exploration. Journal of Applied Logic 4(4), 470–504 (2006)
3. Buchberger, B.: Theory Exploration Versus Theorem Proving. In: Proceedings of the Calculemus 1999 Workshop. Electronic Notes in Theoretical Computer Science, vol. 23(3). Elsevier, Amsterdam (1999)
4. de Clippel, G., Serrano, R.: Bargaining, coalitions and externalities: A comment on Maskin, `http://ssrn.com/abstract=1304712`
5. Deng, X., Papadimitriou, C.H.: On the complexity of cooperative solution concepts. Mathematics of Operations Research 19(2), 257–266 (1994)
6. Grandi, U., Endriss, U.: First-Order Logic Formalisation of Arrow's Theorem. In: He, X., Horty, J., Pacuit, E. (eds.) LORI 2009. LNCS, vol. 5834, pp. 133–146. Springer, Heidelberg (2009)
7. Ireland, A., Bundy, A.: Productive Use of Failure in Inductive Proof. Journal of Automated Reasoning 16(1), 79–111 (1995)
8. Jordan, J.S.: Pillage and property. Journal of Economic Theory 131(1), 26–44 (2006)
9. Jordan, J.S., Obadia, D.: Stable Sets in Majority Pillage Games, mimeo (November 2004)

10. Kerber, M., Rowat, C.: Stable Sets in Three Agent Pillage Games, Department of Economics Discussion Paper, University of Birmingham, 09-07 (June 2009), http://papers.ssrn.com/sol3/papers.cfm?abstract_id=1429326
11. Lakatos, I.: Proofs and Refutations. Cambridge University Press, Cambridge (1976)
12. Lucas, W.F.: A game with no solution. Bulletin of the American Mathematical Society 74(2), 237–239 (1968)
13. Maskin, E.: Bargaining, Coalitions and Externalities. mimeo (June 2003)
14. McKelvey, R.D., McLennan, A.M., Turocy, T.L.: Gambit: Software Tools for Game Theory, Version 0.2010.09.01. mimeo (2010), http://www.gambit-project.org
15. Nipkow, T.: Social Choice Theory in HOL: Arrow and Gibbard-Satterthwaite. Journal of Automated Reasoning 43, 289–304 (2009)
16. Vestergaard, R., Lescanne, P., Ono, H.: The Inductive and Modal Proof Theory of Aumann's Theorem on Rationality. mimeo (2006)
17. von Neumann, J., Morgenstern, O.: Theory of Games and Economic Behavior, 1st edn. Princeton University Press, Princeton (1944)
18. Wiedijk, F.: Formalizing Arrow's Theorem. Sādhanā 34(1), 193–220 (2009)
19. Windsteiger, W., Buchberger, B., Rosenkranz, M.: Theorema. In: Wiedijk, F. (ed.) The Seventeen Provers of the World. LNCS (LNAI), vol. 3600, pp. 96–107. Springer, Heidelberg (2006)
20. Windsteiger, W.: An Automated Prover for Zermelo-Fraenkel Set Theory in Theorema. JSC 41(3-4), 435–470 (2006)

A Qualitative Comparison of the Suitability of Four Theorem Provers for Basic Auction Theory*

Christoph Lange[1], Marco B. Caminati[2], Manfred Kerber[1], Till Mossakowski[3], Colin Rowat[4], Makarius Wenzel[5], and Wolfgang Windsteiger[6]

[1] Computer Science, University of Birmingham, UK
[2] http://caminati.net.tf, Italy
[3] University of Bremen and DFKI GmbH Bremen, Germany
[4] Economics, University of Birmingham, UK
[5] Univ. Paris-Sud, Laboratoire LRI, UMR8623, Orsay, F-91405, France
[6] RISC, Johannes Kepler University Linz (JKU), Austria
http://www.cs.bham.ac.uk/research/projects/formare/code/auction-theory/

Abstract. Novel auction schemes are constantly being designed. Their design has significant consequences for the allocation of goods and the revenues generated. But how to tell whether a new design has the desired properties, such as efficiency, i.e. allocating goods to those bidders who value them most? We say: by formal, machine-checked proofs. We investigated the suitability of the Isabelle, Theorema, Mizar, and Hets/CASL/TPTP theorem provers for reproducing a key result of auction theory: Vickrey's 1961 theorem on the properties of second-price auctions. Based on our formalisation experience, taking an auction designer's perspective, we give recommendations on what system to use for formalising auctions, and outline further steps towards a complete auction theory toolbox.

1 Motivation: Why Formalise Auction Theory?

Auctions are a widely used mechanism for allocating goods and services[1], perhaps second in importance only to markets. They are used to allocate electromagnetic spectrum, airplane landing slots, oil fields, bankrupt firms, works of art, eBay items, and to establish exchange rates, treasury bill yields, and stock exchange opening prices. Novel auction schemes are constantly being designed, aiming to maximise the auctioneer's revenue, foster competition in subsequent markets, and to efficiently allocate resources.

Auction design can have significant consequences. Klemperer attributed the low revenues gained in some government auctions of the 3G radio spectrum in

* This work has been supported by EPSRC grant EP/J007498/1. We would like to thank Peter Cramton and Elizabeth Baldwin for sharing their auction designer's point, and Christian Maeder for his recent improvements to Hets.

[1] For the US, the National Auctioneers Association reported $268.5 billion for 2008 [2].

J. Carette et al. (Eds.): CICM 2013, LNAI 7961, pp. 200–215, 2013.

2000 (€20 per capita vs. €600 in other countries) to bad design [18]. Design practice outstrips theory, especially for complex modern auctions such as combinatorial ones, which accept bids on subsets of items (e.g. collections of spectrum). Designing a revenue-maximising auction is *NP*-complete [6] even with a single bidder. Important auctions often run 'in the wild' with few formal results [19]. We aim at convincing auction designers that investing into formalisation pays off with machine-checked proofs and a deeper understanding of the theory. To this end, we want to provide them with a toolbox of basic auction theory formalisations, on top of which they can formalise and verify their own auction designs – which typically combine standard building blocks, e.g. an ascending auction converting to a sealed-bid auction when the number of remaining bidders equals the number of items available. Given the ubiquity of specialist support across a range of service sectors, we conjecture that auction designers might be supported by formalisation experts, creating a niche for specially trained experts at the interface of the core mechanised reasoning community and auction designers.

Our ForMaRE project (formal mathematical reasoning in economics [22]) seeks to increase confidence in economics' theoretical results, to aid in discovering new results, and to foster interest in formal methods within economics. To formal methods, we seek to contribute new challenge problems and user experience feedback from new audiences. Auctions are representative of practically relevant fields of economics that have hardly been formalised so far.[2] Economics has been formalised before [15], particularly social choice theory (cf. §5 and [10]) and game theory (cf. [37] and our own work [16]). However, none of these formalisations involved economists. Formalising (mathematical) theories and applying mechanised reasoning tools remain novel to economics.[3]

§2 establishes requirements for the Auction Theory Toolbox (ATT); §3 explains our approach to building it. §4 is our main contribution: a qualitative comparison of how well four different theorem provers satisfy our requirements. §5 reviews related work, and §6 concludes and provides an outlook.

2 Requirements for an Auction Theory Toolbox

Conversations with auction designers established ATT requirements as follows:

D1. Formalise ready-to-use basic auction concepts, including their definitions and essential properties.

D2. Allow for extension and application to custom-designed auctions without requiring expert knowledge of the underlying mechanised reasoning system.

From a computer scientist's technical perspective, these translate to:

[2] Even code verification is typically not considered, although Leese, who worked on the UK's spectrum auctions, has called for auction software to be added to the Verified Software Repository at `http://vsr.sourceforge.net` [47].

[3] There is a field 'computational economics'; however, it is mainly concerned with the *numerical* computation of solutions or simulations (cf., e.g., [13]).

C1. Identify the right language to formalise auction theory. This language should (a) be sufficiently expressive for concisely capturing complex concepts, while supporting efficient proofs for the majority of problems, (b) be learnable for economists used to mathematical textbook notation, and (c) provide libraries of the mathematical foundations underlying auctions.

C2. Identify a mechanised reasoning system (a) that assists with cost-effective development of formalisations, (b) that facilitates reuse of formalisations already existing in the toolbox, (c) that creates comprehensible output to help users understand, e.g., why a proof attempt failed, or what knowledge was used in proving a goal, and (d) whose community is supportive towards users with little specific technical and theoretical background.

Note the conflicts of interest: a single language might not meet requirement C1a, and if it did, it might not be supported by a user-friendly system.

3 Approach to Building the Auction Theory Toolbox

To avoid a chicken-and-egg problem, we identify relevant domain problems in parallel to identifying languages and systems suitable for formalisation.

3.1 The Domain Problem: Vickrey's Theorem and Beyond

We started with Vickrey's 1961 theorem on the properties of second-price auctions of a single, indivisible good, whose bidders' private values are not publicly known. Each participant submits a sealed bid; one of the highest bidders wins, and pays the highest *remaining* bid; the losers pay nothing. Vickrey proved that 'truth-telling' – submitting a bid equal to one's actual valuation of the good – was a *weakly dominant* strategy, i.e. that no bidder can do strictly better by bidding above or below their valuation *whatever* the other bidders do. Thus, the auction is also *efficient*, allocating the item to the bidder with the highest valuation. Bidders only have to know their own valuations; in particular they need no information about others' valuations or the distributions these are drawn from.

As variants of Vickrey auctions are widely used (e.g. by eBay, Google and Yahoo! [45]), this formalisation will enable us to prove properties of contemporary auctions as well. The underlying theory is straightforward to understand even for non-economists and can be formalised with reasonable effort. Finally, formalising Vickrey provides a good introduction for domain experts to mechanised reasoning technology by serving as a small, self-contained showcase of a widely known result, helping to build trust in this new technology.

Maskin collected 13 theorems, including Vickrey's, in a review [24] of an influential auction theory textbook [25]. This sets the roadmap for building the ATT – a collaborative effort, to which we welcome community contributions [23].

3.2 Paper Elaboration to Prepare the Machine Formalisation

To prepare the machine formalisation, we refined the original paper source, aware that current mechanised reasoning systems typically require much more explicit

statements than commonly found on paper: automated provers must find proofs without running out of search space, whereas proof checkers require proofs at a certain level of detail, which in turn requires detailed statements. Maskin states Vickrey's theorem in two sentences and proves it in another six sentences [24, Proposition 1].[4] Our elaboration uses eight definitions specific to the domain problem plus an auxiliary one about maximum components of vectors, as follows:

$N = \{1, \ldots, n\}$ is a set of *participants*, often indexed by i. An *allocation* is a vector $\boldsymbol{x} \in \{0,1\}^n$ where $x_i = 1$ denotes participant i's award of the indivisible good to be auctioned (i.e. 'i wins'), and $x_j = 0$ otherwise. An *outcome* $(\boldsymbol{x}, \boldsymbol{p})$ specifies an allocation and a vector of payments, $\boldsymbol{p} \in \mathbb{R}^n$, made by each participant i. Participant i's *payoff* is $u_i \equiv v_i x_i - p_i$, where $v_i \in \mathbb{R}_+$ is i's valuation of the good. A *strategy profile* is a vector $\boldsymbol{b} \in \mathbb{R}^n$, where $b_i \geq 0$ is called i's *bid*.[5] For an n-vector $\boldsymbol{y} = (y_1, \ldots, y_n) \in \mathbb{R}^n$, let $\overline{y} \equiv \max_{j \in N} y_j$ and $\overline{y}_{-i} \equiv \max_{j \in N \setminus \{i\}} y_j$.

Definition 1 (Second-Price Auction). *Given* $M \equiv \left\{i \in N : b_i = \overline{b}\right\}$, *a* second-price auction *is an outcome* $(\boldsymbol{x}, \boldsymbol{p})$ *satisfying:*

1. $\forall j \in N \backslash M, x_j = p_j = 0$*; and*
2. *for one*[6] $i \in M$, $x_i = 1$ *and* $p_i = \overline{b}_{-i}$*, while,* $\forall j \in M \setminus \{i\}, x_j = p_j = 0$.

Definition 2 (Efficiency). *An efficient auction maximises* $\sum_{i \in N} v_i x_i$ *for a given* v*, i.e., for a single good,* $x_i = 1 \Rightarrow v_i = \overline{v}$.

Definition 3 (Weakly Dominant Strategy). *Given some auction, a strategy profile* $\boldsymbol{b}$ *supports an* equilibrium in weakly dominant strategies *if, for each* $i \in N$ *and any* $\hat{\boldsymbol{b}} \in \mathbb{R}^n$ *with* $\hat{b}_i \neq b_i$, $u_i\left(\hat{b}_1, \ldots, \hat{b}_{i-1}, b_i, \hat{b}_{i+1}, \ldots, \hat{b}_n\right) \geq u_i\left(\hat{\boldsymbol{b}}\right)$.[7] *I.e., whatever others do, i will not be better off by deviating from the original bid* b_i.

Theorem 1 (Vickrey 1961; Milgrom 2.1). *In a second-price auction, the strategy profile* $\boldsymbol{b} = \boldsymbol{v}$ *supports an equilibrium in weakly dominant strategies. Furthermore, the auction is efficient.*

The attempt to be close to a paper formalisation may introduce artefacts that unnecessarily complicate machine formalisation. E.g., the contiguous numeric participant indexing is merely a convention: formally any relation between participants' valuation, bid, allocation, and payment vectors suffices. Similarly, the product $v_i x_i$ recalls the general divisible good case ($x_i \in [0,1]$) and works around the lack of an easy and compact 'if–then–else' textbook notation.[8]

[4] The high level of Maskin's text is owed to its summative nature. Original proofs in auction theory are typically more thorough.

[5] This simplification is sufficient for proving the theorem. More precisely, all participants know that each v_i is an independent realisation of a random variable with distribution density f. A participant's *strategy* is a mapping g_i such that $b_i = g_i(v_i, f)$.

[6] When running an auction in practice, this i may be selected randomly, but this circumstance does not matter for the proof of Vickrey's theorem.

[7] The notation $u_i(\boldsymbol{b})$ is standard in economics but formally misleading. A more careful notation is $u_i(x_i, v_i, p_i)$, where x_i and p_i depend on $\boldsymbol{b}$ and the auction type.

[8] Case distinctions with curly braces consume at least two lines.

Proof. Suppose participant i bids $b_i = v_i$, whatever $\hat{b}_j$ the others bid. Let $\hat{\boldsymbol{b}}^{i \leftarrow v}$ abbreviate the overall vector $(\hat{b}_1, \ldots, \hat{b}_{i-1}, v_i, \hat{b}_{i+1}, \ldots, \hat{b}_n)$. There are two cases[9]:

1. i wins. This implies $b_i = v_i = \hat{\boldsymbol{b}}^{i \leftarrow v}$, $p_i = \overline{\hat{\boldsymbol{b}}^{i \leftarrow v}}_{-i}$, and $u_i(\hat{\boldsymbol{b}}^{i \leftarrow v}) = v_i - p_i = \hat{\boldsymbol{b}}^{i \leftarrow v}_i - \overline{\hat{\boldsymbol{b}}^{i \leftarrow v}}_{-i} \geq 0$. Now consider i submitting an arbitrary bid $\hat{b}_i \neq b_i$, i.e. assume an overall bid vector $\hat{\boldsymbol{b}}$. This has two sub-cases:
   (a) i wins with the other bid, i.e. $u_i(\hat{\boldsymbol{b}}) = u_i(\hat{\boldsymbol{b}}^{i \leftarrow v})$, as the second highest bid has not changed.
   (b) i loses with the other bid, i.e. $u_i(\hat{\boldsymbol{b}}) = 0 \leq u_i(\hat{\boldsymbol{b}}^{i \leftarrow v})$.
2. i loses. This implies $p_i = 0$, $u_i(\hat{\boldsymbol{b}}^{i \leftarrow v}) = 0$, and $b_i \leq \overline{\hat{\boldsymbol{b}}^{i \leftarrow v}}_{-i}$; otherwise i would have won. This yields again two cases for i's alternative bid :
   (a) i wins, i.e. $u_i(\hat{\boldsymbol{b}}) = v_i - \overline{\hat{\boldsymbol{b}}}_{-i} = b_i - \overline{\hat{\boldsymbol{b}}^{i \leftarrow v}}_{-i} \leq 0 = u_i(\hat{\boldsymbol{b}}^{i \leftarrow v})$.
   (b) i loses, i.e. $u_i(\hat{\boldsymbol{b}}) = 0 = u_i(\hat{\boldsymbol{b}}^{i \leftarrow v})$.

By analogy for all i, $\boldsymbol{b} = \boldsymbol{v}$ supports an equilibrium in weakly dominant strategies. Efficiency is immediate: the highest bidder has the highest valuation. □

3.3 Choosing Language and System

In terms of *logic*, it is not immediately obvious whether Vickrey's theorem is inherently higher-order. Defining the maximum operator on arbitrarily sized finite sets of real-valued bids and proving its essential properties requires induction and thus exceeds first-order logic (FOL): similarly for the finiteness of a set[10] and a formalisation of real numbers.[11] However, if one takes real vectors and a maximum operation on them for granted, and explicitly requires the maximum to exist, FOL suffices to formalise the relevant domain concepts: single good auctions, second-price auctions, and the theorem statement.[12]

In terms of *syntax*, we assume that auction designers will prefer a language that is close to the textbook mathematics they are used to, rather than having a programming language flavour. We assume that at least optional type annotations support intuitive modelling of domain concepts (e.g. an auction as a function that takes bids and returns an allocation and payments) and prevent formalisation mistakes by cheap early checks (cf. [21]).

In terms of *user experience*, we study two paradigms: *automated provers* try, given a theorem and a knowledge base, to automatically find a proof, potentially appealing to our audience if the user just has to push a button (as with model checkers). *Interactive provers* interactively check a proof written by the user, which may be convenient when a paper proof already exists.

[9] Our initial elaboration of Maskin's proof, which distinguishes cases on the basis of participants' bids, resulted in nine leaf cases. Straightforward on paper, we found them tedious to formalise in Isabelle, which triggered the rearrangement shown here.

[10] Finiteness matters: the set $\{b_i = 1 - \frac{1}{i} : i = 1, 2, 3, \ldots\}$ has no maximum.

[11] Real numbers are not usually required for running auctions in *practice*. Even financial exchanges that allow 'sub-pennying' have a minimal discrete quantum of currency.

[12] For instance, our Mizar proof never invokes any second-order *scheme* directly. Two proof steps use the fact that a finite set of numbers includes its maximum, which is proved in the Mizar Mathematical Library (MML) using the induction scheme.

4 Qualitative Comparison of the Languages and Systems

We have formalised Vickrey's theorem in four systems, which differ in logic, syntax and user experience: Isabelle, followed by Mizar, CASL and Theorema. For each system at least one author has in-depth knowledge. The purpose of redoing formalisations from scratch is to understand the specific advantages and disadvantages of the systems and to obtain as idiomatic a formalisation as possible. The formalisations and instructions for using them are available from the ATT homepage [23]. Tab. 1 compares the features of the systems and their languages and shows the state of our work. The following subsections assess the languages and systems w.r.t. the technical requirements C* of §2. Tab. 2 at the end of this section summarises our findings to underpin our final recommendations.

4.1 Level of Detail and Explicitness Required (req. C1a)

All systems required greater detail and explicitness than the paper elaboration of §3.2. The Isabelle formalisation needs 3 additional definitions and 7 auxiliary lemmas. Guiding the automated provers of Theorema and Hets and Mizar's proof checker required similar numbers of auxiliary statements, plus, in Theorema and Hets, further ones to emulate proof steps (cf. §4.2). However, first steps beyond Vickrey's theorem suggests that these auxiliaries make it easier to formalise *further* notions. As our work involved beginners and experts[13], we can only approximately quantify the formalisation effort beyond the paper elaboration. The 'de Bruijn factor' [40], the formalisation size divided by the size of an informal TEX source, measured after stripping comments and *xz* compression, is around 1.5 for all formalisations[14] except Theorema[15]. This observation suggests that machine formalisation is generally still harder than elaboration on paper.

Even while explicit machine formalisation imposes tedious work on the author, it can also prove beneficial. On paper, it was neither immediately obvious that exactly one participant wins a second-price auction, nor that the outcome is a function of the bids. While obvious that at least two participants are required to define the 'second highest bid', the standard literature largely overlooks this, but formalisation forced us to choose whether to allow it (by, e.g., defining $\max \emptyset \equiv 0$) or to explicitly require $n \geq 2$.

4.2 Expressiveness vs. Efficiency (req. C1a)

As discussed in §3.2, we did not strictly take the elaborated paper source as a specification for the formalisation, but wrote idiomatic formalisations. In Isabelle and Mizar, we, e.g., avoided specific intervals $\{1, \ldots, n\}$ as sets of auction

[13] The Mizar formalisation was, e.g., completely written by an expert (Caminati), whereas the Isabelle formalisation was initially written by a first-time user with a general logic background (Lange), then largely rewritten by an expert (Wenzel).

[14] A typical average is 4, but our paper proof is particularly detailed.

[15] Determining a de Bruijn factor for Theorema does not make sense: single keystrokes or clicks may yield complex inputs, Mathematica notebooks store layout and maintenance information, and Theorema caches proofs in the notebook (cf. §4.6).

Table 1. Languages and systems we compared; state of our formalisations

Language	Logic	Prover	User Interface	Licence	Formalisation
Isabelle/HOL 2013 [14]	HOL (simply-typed set theory)	interactive[a]	document-oriented IDE (Isabelle/jEdit [39]) or programmer's text editor (Proof General Emacs [1])	BSD/LGPL/ GPL	complete incl. proof
Theorema 2.0 [46]	FOL + set theory[b]	automated[c]	textbook-style documents, proof management GUI (add-on for Mathematica CAS)	GPL[d]	statements complete, no proof[e]
Mizar 8.1.01 [11]	FOL[f] + set theory	batch verifier	CLI[g]; programmer's text editor (Emacs add-on)	freeware/GPL+ CC-BY-SA[h]	complete incl. proof
CASL/ TPTP[i] [5]	sorted FOL[i]	automated[j]	progr.'s text editor (Emacs add-on), proof mgmt. GUI+ CLI (Hets 0.98[k] [27]), web service (System on TPTP [34])	GPL	complete incl. proof

[a] Isabelle integrates internal and external automated provers.
[b] Theorema actually supports HOL. We, however, just needed FOL besides the built-in sets, tuples, and the max operator.
[c] For each goal, the prover can be configured individually.
[d] Theorema is under GPL but needs the commercial, closed-source Mathematica. Economists tend to be pragmatic about that.
[e] Theorema is in transition to the new 2.0. Its architecture, inference engine, and user interface are fully implemented, but its collection of *inference rules* is still incomplete. Therefore, the proof does not yet work.
[f] *Schemes* permit a limited degree of higher-order reasoning.
[g] The *verifier* produces a list of numerical errors codes and their source file positions. The ancillary utilities *errflag* and *addfmsg* decorate source files with this information, and optionally append terse textual explanations of the relevant error codes.
[h] The Mizar proof checker is closed-source; the MML is free.
[i] Common Algebraic Specification Language. 'CASL/TPTP' denotes our use of CASL as an input language for automated FOL provers (here: SPASS, E, Darwin) using the TPTP [32] exchange language. CASL features some second-order features, e.g. inductive datatypes.
[j] The proof is largely automatic. However, Vickrey's theorem is too complex to for automated proving in one step. Thus, the proof script introduces auxiliary lemmas and selects suitable axioms and provers for proving them. Proof times range from fractions of seconds if the exact list of axioms used is known beforehand to hours if not. However, once a proof is found, the prover can output the list of axioms used and thus speed up subsequent replays of the proof.
[k] Heterogeneous tool set; gives access to a wide range of automated theorem provers. We use FOL provers, most of which share the unsorted TPTP FOF [32] as a common input format. Hets translates CASL to FOF by introducing auxiliary predicates for sorts.

participants: arbitrary (finite) sets of natural numbers simplify the formalisation, and the highest and second highest bids are determined using library set operations. In contrast, Theorema naturally indexes its built-in tuples from 1 to n and allows for restricting quantified variables to such ranges, e.g. $\forall_{i=1,\ldots,n}$.

The CASL formalisation confirms the assumption of §3.3 that FOL suffices for expressing and proving the essence of Vickrey's theorem. For many FOL provers, CASL's (sub)sorts[16] are mere syntactic sugar but allow us to stay close to the domain language, speaking, e.g., of 'valuation vectors', each of which also is a valid 'bid vector'. Note that we have avoided using partial functions (e.g., for modelling out-of-scope vector indices) because of the complex logic translations required for coding them out.

Isabelle and Mizar process the proof in a few seconds on a 2.5 GHz dual-core processor; Hets/TPTP need about an hour; in Theorema it is not yet complete. We used rather weak HOL features, e.g., no synthesisation of functions. Coinciding with earlier, general observations on HOL [8], the low processing time suggests that there is no disadvantage in choosing a rich logic, which allows for expressing relevant concepts (such as maxima of finite sets of real numbers) naturally. Our formalisations' small size (less than 5 K after compression) does not yet warrant a precise quantitative judgement of time efficiency. Particularly for FOL there exist highly optimised automated provers. They are conveniently accessible in Hets, via the System on TPTP [34] web service (accepting TPTP input that Hets can generate), but also from Isabelle/HOL via the Sledgehammer interface (see §4.3). Still, we observed a source of inefficiency in formalising for automated provers: the high share of preconditions with long conjunctions in our CASL formalisation makes it hard for the automated FOL provers to identify applicable axioms. Such conjunctions result from the absence of structured proofs in CASL. This requires, whenever a theorem is too complex for automated proving, to 'emulate' proofs steps via auxiliary lemmas, whose antecedents are conjunctions of all relevant assumptions in the current branch of the proof tree. Performance improvements by guiding provers through the search space can, however, be achieved with the extra effort of grouping frequently occurring conjunctions of assumptions into single abstract predicates, as in the following concrete case for the proof of Vickrey's theorem: $\mathit{spaWithTruthfulOrOtherBid}(n, x, p, v, \hat{\boldsymbol{b}}, i, \boldsymbol{b}) \Leftrightarrow \mathit{secondPriceAuction}(n, x, p) \wedge |v| = |\hat{\boldsymbol{b}}| = n \wedge \mathit{inRange}(n, i) \wedge \hat{\boldsymbol{b}}_i \neq v_i \wedge \boldsymbol{b} = \hat{\boldsymbol{b}}[i \leftarrow v]$.

4.3 Proof Development and Management (req. C2a)

The systems we studied offer different ways of invoking automated provers and keeping track of proof efforts in progress. The 'apparent' difference between automated and interactive theorem proving blurs at a closer look. The interactive prover Isabelle features various automated proof methods; furthermore Sledgehammer gives access to E, SPASS, and TPTP provers. One can configure the facts they should take into account (e.g. local assumptions and conclusions).

[16] TPTP's typed first-order form (TFF [33]) is sorted, but without subsorts. We have not used it, as Hets cannot currently produce it from CASL.

For Mizar, there are also automated external tools (MPTP, MoMM, Miz$\mathbb{AR}$) [31]. Theorema's automated proving workflow is conceptually similar: specifying the knowledge to be used, then configuring the prover.[17] Hets users can select axioms and previously proved theorems to be sent to an automated prover but have little control beyond. Isabelle's prover configuration is editable within the formalisation source. Theorema stores it in hidden fields within the formalisation and exposes it via a dedicated GUI. Configuring proof tools in Hets is separate from the formalisation: the proof management GUI does not currently store settings persistently; however one can write scripts to be processed on the command line.

Just as Isabelle requires complex statements to be proved in multiple steps, involving different proof methods, the automated provers of Theorema[18] and Hets also require guidance by explicit configuration at times, as can be seen from the `*.hpf` proof scripts in our Hets formalisation [23]. Often, a theorem $c : A \Rightarrow C$ was too complex for automated proving, whereas the job could be done by a script that first proved auxiliary lemmas $a : A \Rightarrow B$ and $b : B \Rightarrow C$, possibly with different provers, and then proved c providing only a and b as axioms. This is conceptually the same as in Isabelle but has four significant user experience differences: 1. Each additional 'proof step' has to be stated as a lemma with full assumptions on the left hand side (similar to the example in §4.2), 2. CASL, originally a specification rather than a prover language, does not syntactically distinguish theorems from lemmas, 3. the scripts have to be maintained separately from the formalisation, and 4. a multi-step proof takes many seconds longer, as Hets translates the input theory from CASL to the respective prover's native language before each proof.[19] This gives a clear incentive to eliminate unnecessary proof steps from a CASL formalisation. This experience also influenced our Isabelle formalisation, where writing multi-step proofs is comparatively painless. There, one lemma had a three-step proof, until experiments with the CASL formalisation made us attempt an automated proof. Thus we realised that we could reduce the Isabelle proof to a single step.[20]

Mizar differs by focusing, instead of built-in tactics and automated proof methods, on a natural deduction style which 'tries to "keep a low profile" in its logical foundations' and aims at 'clarity, human readability and closeness to standard mathematical proofs' [38]. Influenced by Mizar, the Isar language ('intelligible semi-automated reasoning') replaced Isabelle's original tactic interface. In the name of its readability focus, Mizar deliberately prevents users from extending the verifier's power [38, §2.1], often forcing them to justify trivial passages. Mizar's *registrations* do allow for custom automation [4]; however, these at times involute exploits often push registrations beyond their intended scope [20] and may result in implicit inferences and less readable proofs.

Particularly in developing the proof of a theorem as complex as Vickrey's top-down, it is useful to defer proofs of lemmas or proof steps, as to use them

[17] For Theorema, a prover is a *collection of inference rules* applied in a certain *strategy*.
[18] This assessment relies on experience with Theorema 1.
[19] This is necessary as, by default, each successful proof adds one theorem to the theory.
[20] As it makes use of one definition and two lemmas, this was not obvious a priori.

in a larger proof without the workaround of temporarily declaring them as axioms. Theorema proofs can use unproved theorems as knowledge. Isabelle's `sorry` keyword creates a fake proof. CASL theorems are formulas with the annotation `%implied`. When imported into a theory, (open) theorems become axioms, and Hets can use them without proof, but the open proof obligation is still visible in the imported theory. Mizar's verifier offers top-down proving for free by marking unaccepted inferences as errors *and then proceeding*. This results in a formal proof *sketch*, 'very close to informal mathematical English' but still close to a fully formalised proof [41]. Furthermore, one can prefix the keyword `proof` with `@` to expressly and silently skip a proof, or disable the verifier on arbitrary code portions using pragmas. Mizar's Emacs mode exposes these as one-touch macros, which speeds up the verification process and improves interaction [38].

4.4 Library Coverage and Searchability (reqs. C1c, C2b)

To a varying degree we have been able to reuse mathematical foundations from the systems' libraries. Isabelle can *find* reusable material by `find_theorems` queries; Sledgehammer helps to extract a sufficient set of lemmas from the library, which is then minimised towards a necessary set. MML Query is a search engine for the MML [3]. CASL's library is searchable as plain text; Theorema's is not.

Theorema has a built-in tuple type, including a maximum operation, we used it to formalise bid vectors. The CASL library provides inductive datatypes such as arrays [29] but no n-argument maximum operation. The Isabelle/HOL library provides a *Max* operation on finite sets, and various Cartesian product types suitable for representing bids. Given Isabelle's functional programming syntax we found it, however, most intuitive to model our own vectors as functions $\mathbb{N} \to \mathbb{R}$ evaluated up to a given n. Wrappers make the set maximum operator work on these vectors and prove the properties required subsequently. Our Mizar formalisation draws on generic relations and functions, which the MML richly covers. Thus, we only had to add a few interfacing lemmas.

4.5 Term Input Syntax (req. C1b)

Conversations with auction designers suggest that they find Theorema's term input syntax most accessible. The two-dimensional notation in Mathematica notebooks is similar to textbook notation, and our target audience is largely familiar with Mathematica. The syntax of Isabelle and CASL is closer to programming languages. Isabelle's functional type syntax $f : A \Rightarrow B \Rightarrow C$ looks less closely related to textbook notation than CASL's $f : A * B \to C$. Isabelle, CASL and Mizar allow for defining custom 'mixfix' operator notations. Isabelle provides rich translation mechanisms beyond that, but the layout remains one-dimensional, e.g. $\forall x \in A.\ B(x)$ instead of Theorema's $\underset{x \in A}{\forall} B[x]$ for bounded quantification. Isabelle Proof General and Isabelle/jEdit approximate textbook notation by Unicode symbols. Isabelle, Mizar and Hets can export LaTeX. Mizar uses ASCII; its lack of binders makes mathematical concepts such as limits and

sums cumbersome to denote [43]. A major reason for us not to cover the TPTP language is its technical, non-extensible ASCII syntax (using, e.g., !/? for ∀/∃).

Theorema, CASL and Mizar support sharing common quantified variables across multiple statements, corresponding to the practice of starting a textbook section even of several axioms like 'let n, the number of participants, be a natural number ≥ 1'. This helps to avoid redundancy but is prone to copy/paste errors. For example, our CASL formalisation has sections with global quantifiers $\forall i, j$ (e.g. to accommodate the maximum and second-price auction definitions of §3.2), but these include axioms that only use i. Literally pasting into this axiom an expression using j does not cause an error, as j is bound in the current scope as well, but changes the semantics of the axiom in a way hard to detect.

4.6 Comprehensibility and Trustability of the Output (req. C2c)

Machine proofs may 'succeed' for unintended reasons, e.g. accidentally stating a tautology such as an implication with an unsatisfiable antecedent. Or they succeed as intended, but the user cannot follow the (automated) deduction. In such situations the prover's *output* is crucial. Isabelle provides tracing facilities for simplification rules and introduction and elimination rules used in standard reasoning steps. Its inference kernel can produce a full record (usually large and unreadable) of the internal reasoning of automated tools via explicit proof terms, e.g. for independent checking. By default the kernel relies on static ML type-discipline to achieve correctness by construction, without explicit proof terms. Theorema's proof data structure captures the entire proof generation according to the rules and strategy selected. It can be displayed as a structured textbook-style proof with configurable verbosity, and visualised as a browsable tree that distinguishes successful from failed branches. Mizar 'just' verifies what the user wrote according to natural deduction rules, hence he is unlikely to doubt the result. On the other hand, for the same reason, Mizar has no way to detect proofs succeeding for unintended reasons, and offers little help to a user clueless about a failing step. A correct Mizar proof can be improved by enhancer utilities [11, §4.6]: some report useful additional information (e.g., unneeded statements referred in a step, unneeded library files, unneeded lemmas); others cut steps that a human might want to see, impacting readability and possibly the original confidence the user had in the proof. Hets uniformly displays the success of a proof and the list of axioms used; however the latter output is only informative with SPASS. Otherwise, the raw technical output of the prover is displayed, which strongly differs across provers. E.g., SPASS uses resolution calculus, which looks different from a textbook proof. Similarly, System on TPTP outputs performance measures and the status of the given problem (e.g. 'Theorem' or 'Unsatisfiable'), but otherwise the raw prover output.

When a proof attempt fails because the statement was wrong, studying a counterexample may help. Isabelle has the Nitpick counterexample finder built in. Hets integrates several ones (Darwin is supported best [28]) and also employs them for consistency checking, as importing a theory whose axioms have no model results in vacuous truth. Both Isabelle and Hets can attempt a proof or

otherwise try to find a counterexample in the same run. Theorema and Mizar do not support counterexamples.

Before proving, all systems check whether the input is syntactically well-formed and well-typed. Isabelle/jEdit performs parsing, type checking and proof processing during editing, and attaches warnings and error messages like modern IDEs. The other systems require the user to explicitly initiate checking. Mizar and Hets check complete files, whereas in Theorema (which only checks syntax), one can individually check each notebook cell (typically containing one to a few statements). Mizar's verifier is particularly error resilient: it seldom aborts before the last input line, thus reporting errors for the whole file.

4.7 Online Community Support and Documentation (req. C2d)

Community support and documentation are major prerequisites for system adoption. We assume that users with little previous mechanised reasoning and formalisation knowledge will seek low-threshold support from tutorial documents or mailing lists rather than attending community meetings – which, in theorem proving, so far focus on scientific/technical aspects rather than applications.

We compare the community sizes, assuming that large communities are responsive even to non-experts: Isabelle is developed at multiple institutions; its user mailing list gets more than 100 posts a month, with over 1000 different authors since 2000. CASL, an international standard, has been subject of hundreds of publications but does not currently have a mailing list. Hets is mainly developed and used within a single institution; its user mailing list receives less than 10 posts a month. Recalling that Hets is an integrative environment, users can also request help from the communities of TPTP (subject of more than 1000 publications, no mailing list) and individual provers. Theorema is developed within a single institution and will not have a mailing list before the 2.0 release. Mizar is developed at one institution by a team that provides dedicated email user assistance: the 'Mizar User Service'. MML grows by 30–60 articles a year, with 241 contributors so far. The mailing list gets around 10 posts a month.

Isabelle and CASL feature comprehensive tutorials and reference manuals, Hets has a user guide, Mizar offers tutorials [26]. Theorema has partial built-in help texts and is documented in a few publications.

5 Related Work

§1 mentioned earlier efforts to *formalise economics*. Particularly Arrow's impossibility theorem, one of the most striking results in theoretical economics, has been a focus for formalisation efforts, including Nipkow's Isabelle and Wiedijk's Mizar formalisation [30, 42]. As in our case (cf. §3.2), they required initial paper elaboration; additionally, it helped them to identify omissions in their source [9]. This source states three alternative proofs, but Tang's/Lin's fourth, induction-based proof, allowed for obtaining insights on the general structure of social choice impossibility results using computer support [36].

Table 2. Performance (as far as results were comparable)

System/ Language	Proof speed	Textbook closeness		Top-down proofs	Library		Output			Community	Documentation	de Bruijn factor
		PI[a]	TI[a]		LC[a]	LS[a]	PO[a]	CE[a]	WF[a]			
	§4.2	§4.3	§4.5	§4.3	§4.4			§4.6		§4.7		§4.1
Isabelle/HOL	++[b]	++	+	++	++	++	○	++	++	++	++	1.3
Theorema	?	n/a[c]	++	++	+	—	++	n/a	–	—	–	n/a
Mizar	++	++	–	++	++	+	○	n/a	++	+	○	1.7
CASL/TPTP	○[d]	–	+	++	+	–	○	+	+	○	+	1.5

[a] PI/TI = proof/term input; LC/LS = library coverage/search; PO = proof output; CE = counterexamples (incl. consistency checks); WF = well-formedness check. [b] scores from very bad (—) to very good (++) [c] fully GUI-based [d] automated provers

The formal verification technique of model checking has been applied to *auctions*. Tadjouddine et al. proved the strategy statement of Vickrey's theorem via two abstractions to reduce the model checker's search space: program slicing to remove variables irrelevant w.r.t. the property, and discretising bid values (e.g. 'higher than someone's valuation v_i') [35]. Our formalisation is, to the best of our knowledge, the first for *theorem provers*; in the more expressive languages it has the comprehensibility advantage of preserving the structure of the original domain problem. From earlier economics formalisation efforts cited above, it differs in its goal to (ultimately) help economists to use formal methods themselves.

Our focus thus lies on *comparing* different provers by full parallel formalisation. Wiedijk compared Isabelle/HOL, Mizar, Theorema, and 14 other provers by general, technical criteria, studying the code resulting from experts formalising a pure mathematics theorem ($\sqrt{2} \notin \mathbb{Q}$), and comparing it to a detailed paper proof [44]. We complement this with the end user's perspective: our observations, e.g., on the closeness of the input syntax to textbook notation or the comprehensibility of the output are general, but we emphasised these criteria as they are important to auction designers. Griffioen's/Huisman's 1998 PVS and Isabelle/HOL comparison is, like Wiedijk's, independent from a specific application but closer to ours in its look at systems' weaknesses from a user's perspective [12]. Like us, they rate proof management and user support, but go into more detail up to the 'time it takes to fix a bug'. Their *findings* on user interfaces have been obsoleted by progress in developing textbook-like proof languages and editors with random access and asynchronous validation.

6 Conclusion and Outlook

Auctions allocate trillions of dollars in goods and services every year, but their design is still 'far less a science than an art' [24]. We aim at making it a science

by enabling auction designers to verify their designs. By parallel formalisation of the first major theorem in a toolbox for basic auction theory (ATT), we have investigated the suitability of four different theorem provers for this job, taking the perspective not only of experienced formalisers but also of our end users. Our contribution is 2×2-fold: 1. to auction designers we provide (a) a growing library to build their formalisations on, and (b) guidelines on what systems to use; 2. to the CICM community we provide (a) challenge problems[21] and (b) user experience feedback from a new audience. This paper focuses on 1b and 2b.

For a concrete application, our findings confirm the widespread intuitions that formalisation benefits from an initial paper elaboration, that the 'automated vs. interactive' distinction proves of little importance in practice, and that no single system satisfies all requirements. For now, our comparison results in Tab. 2 guide auction designers in choosing a system, given their formalisation requirements and experience. The ideal theorem proving environment would feature a *library* as versatile as in Isabelle or Mizar, a *prover* as efficient as those of Isabelle or Mizar, giving *error messages* as informative as in Isabelle/jEdit, further a *proof input language* as close to textbook style as those of Isabelle or Mizar, or an *interface to explore* automated proofs as informative as Theorema's, a *textbook-like term syntax* as Theorema's, an integration of diverse *tools* as in Isabelle or Hets, and a *community* as lively as Isabelle's. We have not yet exploited all strengths of the systems evaluated: maintaining a growing ATT with increasingly complex dependencies will benefit from stronger modularisation, as supported by Isabelle and even more so by the theory graph management of Hets/CASL. Regarding auction *practice*, we are working towards ways to check that formal definitions of auctions are well-defined functions ('for each admissible bid input there is a unique outcome, modulo some randomness'). Given a constructive proof of this property, it should be possible to obtain verified program code that determines the outcome of an auction given the bids. This may work using Isabelle's code generator, but we will also explore provers based on constructive type theory.

Broader conclusions about auction theory require further research. Bidding typically requires forming conjectures of others' beliefs, involving integration over conditional density functions (cf., e.g., Proposition 13 in Maskin's review [24]). We expect that much of the required foundations should already be available in the libraries of Isabelle and Mizar. Maskin limits his review to single good auctions, noting that few general results exist for multi-unit and combinatorial auctions.[22] Such auctions are often more economically critical (e.g. spectrum auctions, monetary policy [19]) but also more complicated. The real challenge for mechanised reasoning will be to demonstrate its use in this domain.[23]

[21] Our problems are not currently challenging systems' performance but the promises of their languages and libraries.

[22] The last two chapters of [25] address multi-unit auctions; multi-unit and combinatorial auctions are the focus of [7].

[23] Even more ambitiously, many results in auction theory are simplified or extended by explicit application of mechanism design; cf. [17].

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. Auctions: The Past, Present and Future, http://realestateauctionglobalnetwork.blogspot.co.uk/2011/11/auctions-past-present-and-future.html
3. Bancerek, G.: Information Retrieval and Rendering with MML Query. In: Borwein, J.M., Farmer, W.M. (eds.) MKM 2006. LNCS (LNAI), vol. 4108, pp. 266–279. Springer, Heidelberg (2006)
4. Caminati, M.B., Rosolini, G.: Custom automations in Mizar. Automated Reasoning 50(2) (2013)
5. CASL, http://informatik.uni-bremen.de/cofi/wiki/index.php/CASL
6. Conitzer, V., Sandholm, T.: Self-interested automated mechanism design and implications for optimal combinatorial auctions. In: Conference on Electronic Commerce. ACM (2004)
7. Cramton, P., Shoham, Y., Steinberg, R. (eds.): Combinatorial auctions. MIT Press (2006)
8. Farmer, W.M.: The seven virtues of simple type theory. Applied Logic 6(3) (2008)
9. Geanakoplos, J.D.: Three brief proofs of Arrow's impossibility theorem. Discussion Paper 1123RRR. Cowles Foundation (2001)
10. Geist, C., Endriss, U.: Automated search for impossibility theorems in social choice theory: ranking sets of objects. Artificial Intelligence Research 40 (2011)
11. Grabowski, A., Korniłowicz, A., Naumowicz, A.: Mizar in a Nutshell. Formalized Reasoning 3(2) (2010)
12. Griffioen, D., Huisman, M.: A comparison of PVS and isabelle/HOL. In: Grundy, J., Newey, M. (eds.) TPHOLs 1998. LNCS, vol. 1479, pp. 123–142. Springer, Heidelberg (1998)
13. Initiative for Computational Economics, http://ice.uchicago.edu
14. Isabelle, http://isabelle.in.tum.de
15. Kerber, M., Lange, C., Rowat, C.: An economist's guide to mechanized reasoning (2012), http://cs.bham.ac.uk/research/projects/formare/
16. Kerber, M., Rowat, C., Windsteiger, W.: Using *Theorema* in the Formalization of Theoretical Economics. In: Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.) Calculemus/MKM 2011. LNCS (LNAI), vol. 6824, pp. 58–73. Springer, Heidelberg (2011)
17. Kirkegaard, R.: A Mechanism Design Approach to Ranking Asymmetric Auctions. Econometrica 80(5) (2012)
18. Klemperer, P.: Auctions: theory and practice. Princeton Univ. Press (2004)
19. Klemperer, P.: The product-mix auction: a new auction design for differentiated goods. European Economic Association Journal 8(2-3) (2010)
20. Korniłowicz, A.: On Rewriting Rules in Mizar. Automated Reasoning 50(2) (2013)
21. Lamport, L., Paulson, L.C.: Should your specification language be typed? ACM TOPLAS 21(3) (1999)
22. Lange, C., Rowat, C., Kerber, M.: The ForMaRE Project – Formal Mathematical Reasoning in Economics. In: Carette, J., Aspinall, D., Lange, C., Sojka, P., Windsteiger, W. (eds.) CICM 2013. LNCS (LNAI), vol. 7961, pp. 330–334. Springer, Heidelberg (2013)
23. Lange, C., et al.: Auction Theory Toolbox (2013), http://cs.bham.ac.uk/research/projects/formare/code/auction-theory/

24. Maskin, E.: The unity of auction theory: Milgrom's master class. Economic Literature 42(4) (2004)
25. Milgrom, P.: Putting auction theory to work. Cambridge Univ. Press (2004)
26. Mizar manuals (2011), http://mizar.org/project/bibliography.html
27. Mossakowski, T.: Hets: the Heterogeneous Tool Set, http://dfki.de/cps/hets
28. Mossakowski, T., Maeder, C., Codescu, M.: Hets User Guide. Tech. rep. Version 0.98. DFKI Bremen (2013), http://informatik.uni-bremen.de/agbkb/forschung/formal_methods/CoFI/hets/UserGuide.pdf
29. Mosses, P.D. (ed.): CASL Reference Manual. LNCS, vol. 2960. Springer, Heidelberg (2004)
30. Nipkow, T.: Social choice theory in HOL: Arrow and Gibbard-Satterthwaite. Automated Reasoning 43(3) (2009)
31. Rudnicki, P., Urban, J., et al.: Escape to ATP for Mizar. In: Workshop Proof eXchange for Theorem Proving (2011)
32. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. Automated Reasoning 43(4) (2009)
33. Sutcliffe, G., Schulz, S., Claessen, K., Baumgartner, P.: The TPTP Typed First-order Form with Arithmetic. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18. LNCS (LNAI), vol. 7180, pp. 406–419. Springer, Heidelberg (2012)
34. System on TPTP, http://cs.miami.edu/~tptp/cgi-bin/SystemOnTPTP
35. Tadjouddine, E.M., Guerin, F., Vasconcelos, W.: Abstracting and Verifying Strategy-Proofness for Auction Mechanisms. In: Baldoni, M., Son, T.C., van Riemsdijk, M.B., Winikoff, M. (eds.) DALT 2008. LNCS (LNAI), vol. 5397, pp. 197–214. Springer, Heidelberg (2009)
36. Tang, P., Lin, F.: Computer-aided proofs of Arrow's and other impossibility theorems. Artificial Intelligence 173(11) (2009)
37. Tang, P., Lin, F.: Discovering theorems in game theory: two-person games with unique pure Nash equilibrium payoffs. Artificial Intelligence 175(14-15) (2011)
38. Urban, J.: MizarMode—an integrated proof assistance tool for the Mizar way of formalizing mathematics. Applied Logic 4(4) (2006)
39. Wenzel, M.: Isabelle/jEdit – a Prover IDE within the PIDE framework. In: Jeuring, J., Campbell, J.A., Carette, J., Dos Reis, G., Sojka, P., Wenzel, M., Sorge, V. (eds.) CICM 2012. LNCS (LNAI), vol. 7362, pp. 468–471. Springer, Heidelberg (2012)
40. Wiedijk, F.: De Bruijn factor, http://cs.ru.nl/~freek/factor/
41. Wiedijk, F.: Formal proof sketches. In: Berardi, S., Coppo, M., Damiani, F. (eds.) TYPES 2003. LNCS, vol. 3085, pp. 378–393. Springer, Heidelberg (2004)
42. Wiedijk, F.: Formalizing Arrow's theorem. Sādhanā 34(1) (2009)
43. Wiedijk, F.: The QED Manifesto Revisited. Studies in Logic, Grammar and Rhetoric 10(23) (2007)
44. Wiedijk, F. (ed.): The Seventeen Provers of the World. LNCS (LNAI), vol. 3600. Springer, Heidelberg (2006)
45. Wikipedia (ed.): Vickrey auction (2012), http://en.wikipedia.org/w/index.php?title=Vickrey_auction&oldid=523230741
46. Windsteiger, W.: Theorema 2.0: A Graphical User Interface for a Mathematical Assistant System. In: UITP Workshop at CICM (2012)
47. Woodcock, J., et al.: Formal method: practice and experience. ACM Computing Surveys 41(4) (2009)

CreaComp: Computer-Supported Experiments and Automated Proving in Learning and Teaching Mathematics[1]

Günther Mayrhofer, Susanne Saminger, Wolfgang Windsteiger
JKU Linz, Austria
Institut für Algebra, guenther.mayrhofer@students.jku.at
Institut für Wissensbasierte Mathematische Systeme, susanne.saminger@jku.at
Institut für Symbolisches Rechnen, wolfgang.windsteiger@risc.uni-linz.ac.at

Abstract

We present an environment for learning and teaching mathematics that aims at inspiring the creative potential of students by enabling the learners to perform various kinds of interactive experiments during their learning process. Computer interactions are both of visual and purely formal mathematical nature, where the computer-algebra system *Mathematica* powers the visualization of mathematical concepts and the tools provided by the theorem proving system *Theorema* are used for the formal counterparts.

1. Introduction

Both in classroom and in scenarios of distance learning of mathematics the use of computer-algebra systems has become more and more popular. Recently computer-support focuses on *(symbolic) computations*, e.g. calculating limits, derivatives, integrals, and *visualization*, e.g. plotting real-valued sequences or functions: the availability of powerful algorithms allows to perform calculations even in situations when the method would require a tremendous computational effort when executed "by hand" or when the algorithms are not known to the students at a certain level of education. Since modern symbolic computation systems are powerful enough to carry out all computations done in high-school and undergraduate mathematics, the question of which methods should be taught to students and for which tasks we rely on the help of the computer is of utmost importance. There is no absolute answer to this and the "White-Box Black-Box principle", see [2], usually serves as the didactic guideline, by which, depending on the didactical goals certain methods are taught in detail in one phase of education (the white-box phase) whereas they can be applied as black-boxes, e.g. by using a computer, in later phases.

On the other hand, most of the available electronic learning material for mathematics only rarely focus on computer-support for acquiring such crucial mathematical skills as "exact formulation of mathematical properties" and "rigorously proving mathematical properties correct", for exceptions see e.g. [1,7].

In this paper we present the CREACOMP project, whose main goal is to develop electronic course material for self-study and also for use in classroom. In its current state, CREACOMP comprises approximately 15 (only loosely connected) learning units on an undergraduate level, such as e.g. equivalence relations, polynomial interpolation, or Markov processes. The computer-aid provided in CREACOMP units follows the didactical guidelines set up in the MEETMATH project, see [6], and it promotes an approach of self-paced learning that has been centered around *gaining insight into mathematical concepts through computational and graphical interaction*. In addition to computational and graphical interaction the CREACOMP approach now adds *formal reasoning* as a third important component by integrating the *Theorema* system, see [3,4,5]. *Theorema* is designed

[1]This work is done within the project "CreaComp: E-Schulung von Kreativitaet und Problemlösekompetenz" at the Johannes Kepler University of Linz sponsored by the Upper Austrian government.

to become a uniform mathematical computer environment, but in the CREACOMP context, among its many features it mainly provides the mathematical language and the possibility of fully automated generation of mathematical proofs.

The CREACOMP approach aims to combine the experimental approach of discovering mathematics through interactive visualizations and computations with the rigorous approach of proving every claim that is made. By using the *Theorema* system, an experimental flavor can also be given to the formal part, i.e. students can observe with little effort how the *available knowledge influences success or non-success of a proof*, how *tacit assumptions* are often used in human arguments, or how *mathematical theories evolve* from sometimes simple definitions.

We present a case study on the concept of equivalence relations and set partitions, in which we demonstrate the entire bandwidth of computer-support that we envision for modern learning environments for mathematics ranging from "getting first ideas and intuitions" over "checking the validity of first ideas on a wide variety of examples" until "rigorously proving one's own conjectures".

2. Computer-Support in CreaComp Learning Units

The most dangerous scenario of computer-supported mathematics is that the extensive use of computation and visualization may lead to a tradition of "proof by inspecting particular examples". The combination of MEETMATH and *Theorema* in the frame of the CREACOMP-project, therefore, aims at providing computer aid for visualization, computation, but most importantly also for proving. MEETMATH provides a didactical concept, in which the learning phase is structured into phases of *motivation*, *acquisition*, *strengthening*, and *assessment*. During acquisition phases we use computation and visualization in order to fertilize the *intuition* about mathematical objects. The newly acquired knowledge is then formulated rigorously in the *Theorema* language, and the subsequent proofs serve for *strengthening*. At the same time, the use of an automated theorem prover and its possibility to generate human-readable proofs within a few seconds can enhance the *understanding* of mathematical argumentation, and therefore serves the *acquisition*, this time, however, on the learning level of mathematical problem solving techniques. Moreover, we think that by having a machine to give formal proofs of all statements and therefore being forced to fill all gaps in the proofs, the students can better understand the evolution of mathematical theories.

The interplay of MEETMATH and *Theorema* is facilitated by the common underlying *Mathematica* technology, since both make use of the capabilities of the *Mathematica* notebook-frontend. CREACOMP educational units are distributed in the form of *Mathematica* notebooks containing structured text, inlined or displayed mathematical formulae, graphics, and all sorts of *Mathematica*/*Theorema* input and output. Formal entities such as definitions or theorems, which we later want to use in computations or proofs, are written as *Theorema* input cells. Input cells differ in layout from the surrounding text blocks so that they can easily be recognized as active content. These cells must be evaluated in order for their content to be accessible in the *Mathematica* kernel later. Note, however, that *Theorema* input is quite different from usual *Mathematica* input! *Theorema* understands most of the common mathematical notation, notably all sorts of quantifiers written in appealing two-dimensional form and, thus, *Theorema* input hardly differs from inlined mathematical formulae in the text. Although *Mathematica* and *Theorema* provide palettes and keyboard shortcuts for inputting two-dimensional expressions, *Theorema* input is always prepared in advance and the users are usually not required to type *Theorema* formulae.

Furthermore we provide visualization commands for certain mathematical objects. In addition to static plots we try to involve the students in actively exploring mathematical properties by providing *interactive visualizations* based on *Mathematica*'s GUIKit, a toolbox supporting the implementation of Java applications fed with *Mathematica* data. The availability of an interactive experiment is always indicated by *interaction buttons* in the running text. Buttons appear as gray

boxed text and clicking a button calls its associated button-function, in which we start the interactive experiment, usually a Java application implemented in the *Mathematica* programming language and linked to *Mathematica* through the GUIKit. The application contains certain interactive elements such as text fields, buttons, checkboxes, or sliders, and some graphics that refreshes on any user interaction given through these active elements. The surrounding text and the text on the button roughly explains the associated experiment. In addition, all experimental tools are equipped with an online help that explains the possible interactions and, as a side-effect, it is meant to lead the students' experiments into a "reasonable direction" so that they might conjecture "relevant knowledge". Students have the freedom in exploiting the interactive tools in arbitrary manner, but it is important to provide them also some guidelines in what to try and what to observe, otherwise there is the danger that they get lost if they deviate from the intended track through the unit.

Finally, we encourage students to also prove their conjectures after their experiments. In this stage they may use the *Theorema* provers both in interactive or in fully automated mode. When it comes to automatically proving a conjecture expressed in *Theorema* language we provide a CREACOMP *prove panel* in order to hide the concrete call of the respective *Theorema* prover from the user. The prove panel serves as the uniform interface to *Theorema*'s automated provers, and it is made up of several buttons on a gray area spanning the entire width of the notebook. The prove-button on the left has a call to a *Theorema* prove method with appropriate parameters associated to its "button-pressed"-event. Every *Theorema* prover needs the knowledge base to be used in the proof as a parameter, thus, before actually starting the proof the user must compose the knowledge base in an interactive dialog. This so-called knowledge base composer displays the formula to be proven and it lists all definitions, propositions, theorems, etc. available at this stage. The user simply selects by mouse-click and sends the knowledge base to the prover by pressing the "Prove"-button in the dialog's bottom-right corner. Pressing the "Hint"-button in the bottom-left corner selects just the "appropriate" portion of knowledge for a successful proof. The appropriate knowledge for a certain proof cannot be detected automatically, the developer of the unit needs to hide this information within the prove panel so that the knowledge base composer can access it from there. Furthermore, the prove panel contains a button that allows to view a pre-generated successful proof. Both pre-generated and live-generated proofs appear in a separate window, and they come in human-readable format and explain each proof step in natural language. Moreover, *Theorema* proofs have some remarkable features: All formula references are active button elements, which will display the referenced formula in a small separate window, proof goals and assumptions can easily be distinguished by color, and the structure of the proof tree is reflected by the nested cell structure of the proof notebook, so that entire proof branches can be collapsed by a single mouse-click.

In all facets of the CREACOMP user interface we make heavy use of interactive notebook elements like buttons and hyperlinks in order to prevent students from struggling with unfamiliar input syntax. Due to the use of Java and *Mathematica*'s GUIKit, user experiments are mainly based on common modern user interface actions such as selecting items by clicking radio-buttons or checkboxes, opening dialogs by button-click, etc. instead of the usual *Mathematica* command interface.

3. Case Study: Equivalence Relations and Set Partitions

We try to illustrate the flavor of computer-support given in a CREACOMP unit in the example chapter on "equivalence relations and set partitions", in which we span from the definition of an *equivalence relation* in set theory and the definition of the *relation induced by a set of sets* to the main theorem that the relation induced by the factor set of an equivalence relation R is equal to R.

The unit starts by some motivation why we want to study equivalence relations, including hyperlinks to related units, and a summary of *Mathematica*/*Theorema* commands needed in this

section. Firstly, the notion of a binary relation R on some universe A is defined in the *Theorema* language, immediately followed by the definitions of the main concepts leading to equivalence relations, namely *reflexivity*, *symmetry*, and *transitivity*. An interaction button opens the first interaction possibility, where the user can choose arbitrary relations to be visualized by their adjacency matrix in form of a chessboard-like raster array. The relations can be user-defined, or random reflexive/symmetric/transitive relations can be generated. A graphical interpretation of some properties of relations can be seen in these experiments.

Next the concept of the *class* of x w.r.t. a relation R on a universe A is introduced. Using *Theorema*, this writes as[2]

Definition["class",any[x,A,R], class[x,A,R]:=$\{a \in A | \langle a,x \rangle \in R\}$].

Although its appearance comes close to how a definition would appear in a textbook, this *Theorema* definition is *active input* and after evaluating the input cell, the definition can be referred to in this session as Definition["class"]. Then the user is confronted with a visualization tool for classes. The experiment in this tool is similar to the above mentioned, only in addition to the adjacency matrix it shows an additional graphics with a small disk for each element of the universe arranged in a circle with a unique color assigned to each of them. An elements color is shown in the color of its label, if an element y belongs to the class of x, then y's disk is colored in x's color, if y belongs to more than one class, then its disk is gray. Via checkboxes the user can mark the elements, whose classes are to be visualized.

The notions reflexivity, symmetry, and transitivity are then explored in their relationship to classes. Relations having the respective properties are first visualized using the interactive tool just described, then conjectures are formulated in the *Theorema* language, e.g. that for a reflexive relation the union of all classes is equal to the universe, or that for a symmetric transitive relation the classes of two elements x and y coincide as soon as x belongs to the class of y.

Of course, not all conjectures are always true. Some are false in general, but they may be valid in special cases. For example, after inspecting relations in the visualization tool some students might believe, for instance, that every element is member in its own class After trying the proof they would realize that the prover fails to prove this proposition. In general, failure of a *Theorema* proof can have various reasons: the provided knowledge is insufficient or the proposition as stated is not provable, maybe not even true! This is just what we want to emphasize in teaching mathematics, and conventional learning material does not provide room for this experience. Finally, failure can also be due to an inappropriate proving method or inappropriate parameters for the method, but we eliminate these sources by packing the call to the prover into the prove panel.

Theorema's possibility to inspect failing proof attempts comes very handy at this point, so students can investigate, which part of the proof led to failure. In the example above, they detect that the prover closes all branches successfully only $\langle x,x \rangle \in R$ needs to be proven for arbitrary x and R. Thorough inspection of the available knowledge at that stage shows that only $x \in A$ is known and the student might learn that the propsition as stated *cannot be proven* unless more is known about R. Thus, they have to add more side-conditions to the proposition. In *Theorema* this can be done using the with[...] expression. At this place, the unit now contains a proposition with a placeholder, where the students can insert the additional side-condition. Now a user can try different conditions in order to succeed in proving the proposition. Of course, in order to succeed with the above proof, R must have the property that $\langle x,x \rangle \in R$ is true for all $x \in A$, i.e. R must be reflexive on A. Hence, we must put this side-condition and can then successfully prove the adapted proposition. This proof is not particularly difficult, still the formal rigor in which it is carried out is instructive for students.

---

[2]The definition in *Theorema* format can use arbitrary mathematical syntax and special symbols. In the concrete case, we use currying and write R and A as subscripts to "class" hence "class of ... w.r.t. R in A" having only one parameter x.

The remaining content of this unit explains some properties of sets of sets. In particular, the students can learn about the factor set of a relation, i.e. the set of all classes, and which properties a set of sets make it a "partition". Moreover, the concept of an "induced relation" of a set of sets is introduced, namely the set of all pairs, for which there is a set containing its both components. In each step the procedure is similar: first *introduce* new objects or properties, then *visualize* in order to gain insights, *guess and propose* conjectures, *formalize* the conjectures precisely, and *prove* them automatically with *Theorema*. The key observations are then:

- if R is an equivalence relation on A then the factor set of R is a partition of A and its induced relation equals R.
- if P is a partition of A then the induced relation of P is an equivalence relation and its factor set equals P.

All these theorems including also all auxiliary lemmata necessary for compact proofs of the main statements are proved fully automatically within this learning unit.

4. Conclusion and Future Work

CREACOMPis work in progress. Therefore we do not have results on evaluation of the units in classroom yet. Further work will go into computer-supported assessment, which has already been implemented in the frame of MEETMATH, see [6]. Assessment is heavily based on randomly generated test exercises based on example patterns, where the power of a symbolic computation system in the background is essential for checking correctness of user solutions. Also, the use of *Theorema* provers for checking user answers can be investigated.

5. References

1. R.B. Andrews, C.E. Brown, F.Pfenning, M.Bishop, S.Issar, and H.Xi. ETPS: A System to Help Students Write Formal Proofs. *Journal of Automated Reasoning*, 32:75–92, 2004.
2. B.Buchberger. Should Students Learn Integration Rules? *ACM SIGSAM Bulletin*, 24(1):10–17, January 1990.
3. B.Buchberger, A.Craciun, T.Jebelean, L.Kovacs, T.Kutsia, K.Nakagawa, F.Piroi, N.Popov, J.Robu, M.Rosenkranz, and W.Windsteiger. Theorema: Towards Computer-Aided Mathematical Theory Exploration. *Journal of Applied Logic*, 2005. To appear.
4. B.Buchberger, C.Dupre, T.Jebelean, F.Kriftner, K.Nakagawa, D.Vasaru, and W.Windsteiger. The Theorema Project: A Progress Report. In M.Kerber and M.Kohlhase, editors, *Symbolic Computation and Automated Reasoning (Proceedings of CALCULEMUS 2000, Symposium on the Integration of Symbolic Computation and Mechanized Reasoning)*, pages 98–113. St. Andrews, Scotland, Copyright: A.K. Peters, Natick, Massachusetts, 6-7 August 2000.
5. B.Buchberger, T.Jebelean, F.Kriftner, M.Marin, E.Tomuta, and D.Vasaru. A Survey of the Theorema project. In W.Kuechlin, editor, *Proceedings of ISSAC'97 (International Symposium on Symbolic and Algebraic Computation, Maui, Hawaii, July 21-23, 1997), ACM Press 1997.*, pages 384–391, 1997.
6. S.Saminger. MeetMATH — visualizations and animations in a didactic framework. In M.Borovcnik and H.Kautschitsch, editors, *Technology in Mathematics Teaching (Special groups and working groups). Proceedings of ICTMT 5, Klagenfurt (Austria)*, volume 26 of *Schriftenreihe Didaktik der Mathematik*, pages 217–222, Wien, 2002. цbv & hpt Verlagsgesellschaft.
7. R.Sommer and G.Nuckols. A Proof Environment for Teaching Mathematics. *Journal of Automated Reasoning*, 32:227–258, 2004

CreaComp: Experimental Formal Mathematics for the Classroom*

Günther Mayrhofer[1] and Susanne Saminger[2] and Wolfgang Windsteiger[3]

[1] *Institut für Algebra, guenther.mayrhofer@students.jku.at*
[2] *Institut für Wissensbasierte Mathematische Systeme, susanne.saminger@jku.at*
[3] *Institut für Symbolisches Rechnen, wolfgang.windsteiger@risc.uni-linz.ac.at*
JKU Linz, Altenbergerstraße 69, A-4040 Linz, Austria

CreaComp provides an electronic environment for learning and teaching mathematics that aims at inspiring the creative potential of students. During their learning process, students are encouraged to engage themselves in various kinds of interactive experiments, both of visual and purely formal mathematical nature. The computer-algebra system *Mathematica* powers the visualization of mathematical concepts and the tools provided by the theorem proving system *Theorema* are used for the formal counterparts. We present a case study on the concept of equivalence relations and set partitions, in which we demonstrate the entire bandwidth of computer-support that we envision for modern learning environments for mathematics.

Keywords: Automated Theorem Proving, Computer-Algebra Systems, Maths Education

1. Introduction

Both in classroom and in scenarios of distance learning of mathematics, the use of computer-algebra systems has become more and more popular. Until now, computer-support focuses on *(symbolic) computations*, e.g. calculating limits, derivatives, integrals, polynomial and matrix computations, and *visualization*, e.g. plotting real-valued sequences or functions or other mathematical objects in 2D or 3D: the availability of powerful algorithms allows to solve certain problems even in situations when the method would require a tremendous computational effort when executed by hand or when no solution method is known to the students at a certain level of educa-

*This work is done within the project "CreaComp: E-Schulung von Kreativität und Problemlösekompetenz" at the Johannes Kepler University of Linz sponsored by the Upper Austrian government.

tion. Additionally, different visual representations of the computational objects contribute to qualitative data analysis and exploration of non-obvious mathematical relationships. It is clear that modern symbolic computation systems are powerful enough to carry out all computations done in high-school mathematics and most of the computations taught at undergraduate university level. Hence, the questions of which methods should be taught to students in the first place, and for which tasks we rely on the help of the computer, are of utmost importance. There is no absolute answer to this and the "White-Box Black-Box principle", see [3], usually serves as the didactic guideline, by which, depending on the didactical goals, certain methods are taught in detail in one phase of education, the white-box phase, whereas they can be applied as black-boxes in later phases.

On the other hand, most of the available electronic learning material for mathematics only rarely focus on computer-support for acquiring such crucial mathematical skills as "exact formulation of mathematical properties" and "rigorously proving mathematical properties correct", for exceptions see e.g. [1,12]. In any case, computers can support students and teachers, but they can also introduce new obstacles compared to traditional learning/teaching, see e.g. [8].

In this paper we present the CreaComp project, whose main goal is to develop electronic course material for self-study and also for use in classroom. In its current state, CreaComp comprises approximately 15 (only loosely connected) learning units on an undergraduate level, such as e.g. equivalence relations, polynomial interpolation, or Markov processes. The computer-aid provided in CreaComp units follows the didactical guidelines set up in the MeetMath project, see Section 2, and it promotes an approach of self-paced learning that has been centered around "gaining insight into mathematical concepts through computational and graphical interaction". In addition to this, the CreaComp approach now adds *formal reasoning* as a third important component by integrating the *Theorema* system, see [5–7]. *Theorema* is designed to become a uniform environment in which a mathematician gets support during all periods of his/her mathematical occupation. For the CreaComp project, however, *Theorema* mainly provides the mathematical language and the possibility of fully automated or interactive generation of mathematical proofs. Both MeetMath and *Theorema* are based on the well-known computer algebra system *Mathematica*, see [14].

There is often the distinction between *experimental mathematics* and *formal mathematics* and computer-supported teaching/learning is primar-

ily associated with experimental mathematics whereas all proving-related mathematics is predominantly considered to be done "by hand". The discussion is then "experimental mathematics vs. formal mathematics" and the question is to what extent can formal mathematics be supplemented/accompanied/enriched/replaced by experimental mathematics and vice versa, see [2]. The CreaComp approach should be seen much more as "experimental formal mathematics", we aim to combine the experimental approach of discovering mathematics through interactive visualizations and computations with the rigorous approach of proving every claim that is made. By using the *Theorema* system for automatically generating human-readable proofs, an experimental flavor can also be given to the formal part, i.e. students can observe with little effort how the *available knowledge influences success or non-success of a proof*, how *tacit assumptions* are often used in human arguments, or how *mathematical theories evolve* from sometimes simple definitions. Although CreaComp is not intended as an environment for learning how to prove, the white-box nature of *Theorema* proofs may even assist the student in acquiring some proving skills.

In the sequel, we will briefly introduce the constituent components MeetMath and *Theorema* and their combination in Section 2, the main part will be an exemplary case study presenting parts of a CreaComp unit on equivalence relations and set partitions in Section 3. Mathematically, the learning unit on equivalence relations and set partitions starts from

- the *definition of binary relations* and elementary properties such as *reflexivity*, *symmetry*, and *transitivity*,
- then introduces *classes* and *factor sets*,
- proceeds with *set partitions* and *induced relations*,
- develops the theorems that *the factor set of an equivalence forms a set partition* and that *the induced relation of a set partition is an equivalence relation*, and
- finally concludes with the theorems that *building the factor set (of an equivalence relation)* and *building the induced relation (of a set partition)* are *inverse* to each other.

At all stages, interactive visualization tools are provided to illustrate the new concepts and their properties in small and easily comprehensible examples. For all theorems as well as for all auxiliary lemmata required in the proofs, we allow for automated proofs in natural language to be generated interactively by the students, i.e. we provide an interface to *Theorema*-provers that allows the student to generate fully automated proofs.

2. The CreaComp Project: An Overview

2.1. *The Components MeetMath and Theorema*

MeetMath denotes a family of interactive mathematics course-ware based on *Mathematica* equipped with a Java^{TM}-based navigation. The basic course MeetMath@Business&Economy and the didactic concepts for MeetMath course-ware have been initially developed in the framework of the project IMMENSE (for *I*nteractive *M*ultimedia *M*athematics *E*ducation in *N*etworked universities for *S*ocial and *E*conomic sciences) initiated by the Johannes Kepler University Linz already in 1999. For more detailed information about the project and partners see [11].

The development of any course material including electronic supplement demands to focus not only on, possibly new, technology but on the corresponding didactic framework as well. MeetMath course units are structured as "didactical rooms", where different rooms reflect different phases of the learning process. Basically, four different types of didactical rooms have been distinguished:

(i) "Motivation/linking": motivation/linking addresses the students' knowledge about and their attitude to deal with certain content.
(ii) "Acquisition/confrontation": the central new concepts are presented in a complete and carefully paced argumentation. Relevant information, thoughts, and illustrations should be offered in a way that students are able to build up their own ideas and concepts and/or to modify existing, incorrect concepts.
(iii) "Strengthening": these rooms allow to verify and inspect newly developed concepts and ideas. Experiments and interactive elements enable students to stabilize the concepts and allow to identify incorrectly developed concepts by trial and error. Both success and failure are possible and allowed during strengthening.
(iv) "Assessment": assessment is a necessary tool for supervising the learning process. Since students are responsible for the development of their knowledge, they need some tools for measuring the progress and/or for finding out the status quo.

Theorema is a system that intends to bring computer-support during all phases of mathematical activity, such as proving, solving, and computing. The *Theorema* system provides a uniform language and logic, in which many of the above activities can be carried out, see [4]. The overall design principle of the *Theorema* system is to communicate with the user

in mathematical textbook style. The syntax of the language in *both input and output* is close to common mathematical notation, including special mathematical characters and two-dimensional syntax as typically used in mathematics. The *Theorema* language is a version of higher-order predicate logic with pre-defined basic mathematical objects such as numbers, sets, and tuples. The main focus in the development of the *Theorema* system over the past years has been put on the development of various general and special-purpose fully automated theorem provers.

Since *Theorema* is built on top of the well-known *Mathematica* system, we use the *Mathematica* notebook front-end as the user-interface for *Theorema*. When generating mathematical proofs, *Theorema* displays the full proof in a separate notebook document with each proof step explained in natural language. The structure of the proof is reflected in the cell structure of the proof notebook, so that the standard *Mathematica* technology of opening/closing nested cells by mouse-click can be used to collapse entire proof branches. This allows the reader to get an overview over the structure of the proof and then to zoom into parts of the proof by subsequently opening just the relevant cells. As an alternative to fully automated proof generation the *Theorema* system also allows for interactive proving, see e.g. [10]. For details on the *Theorema* system, the language, available provers, and applications of the *Theorema* system, we refer to the introductory papers [5–7]. Notably, an overview on CreaComp as an application of *Theorema* in education has been given in [13].

2.2. *The Combination of MeetMath and Theorema*

The most dangerous scenario of computer-supported mathematics is that the extensive use of computation and visualization leads to a tradition of "proof by inspecting particular examples" instead of "proof by mathematical proving". In our view, examples can and should accompany the development of mathematical content, they can contribute to shaping the students' intuition about mathematics, and the content presented in examples is typically well memorized. However, examples—even if many and well-chosen—can (almost) never substitute a proof of a proposition. It is one of the main goals of this project to highlight the importance of rigorous formal mathematical arguments in all facets of mathematical work, including for instance also software development.

The combination of MeetMath and *Theorema*, which is investigated in the frame of the CreaComp project, therefore aims at providing computer aid for visualization, computation, but most importantly also for

proving. Whereas computation and visualization ought to fertilize the *intuition* about mathematical objects, the proving phase should establish and enhance the *understanding* of mathematical argumentation, in particular that "validity in some examples" does not necessarily always mean "validity in *all cases*". An additional benefit of a theorem proving system as the student's assistant is that the students must think carefully about tacit assumptions they use in their argumentation, because the automated prover forces them to state all usable knowledge explicitly, see also e.g. [9]. Moreover, we think that by having a machine to give formal proofs of all statements and therefore being forced to fill all gaps in the proofs, the students can understand the evolution of mathematical theories much better. We consider this experience, even if maybe not necessary for a pure user of mathematics, as very enlightening for students of mathematics.

The interplay of MeetMath and *Theorema* is facilitated by the common underlying *Mathematica* technology, since both are based on the capabilities of the *Mathematica* notebook-frontend. CreaComp educational units are distributed in the form of *Mathematica* notebooks containing normal text intermixed with formal mathematical texts (definitions, theorems, lemmata, etc.) written in the *Theorema* language. Furthermore, we provide visualizations of the mathematical objects studied in the units. In addition to static plots we involve the students in actively exploring mathematical properties by providing *interactive visualizations* based on *Mathematica*'s GUIKit, a toolbox supporting the implementation of Java applications fed with *Mathematica* data. Finally, we encourage students to also prove their conjectures after their experiments. In this stage, they may use the *Theorema* provers both in interactive or in fully automated mode.

Although the *Mathematica* notebook-frontend is often called a graphical user interface (GUI), *Mathematica* propagates a command-centered interaction pattern. In order to trigger a *Mathematica* computation, a command has to be typed into the notebook and then needs to be evaluated by pressing certain keys. For the CreaComp interface we make heavy use of interactive notebook elements like buttons and hyperlinks in order to prevent students from struggling with unfamiliar input syntax. Students' experiments are mainly based on common modern user interface actions such as selecting items by clicking radio-buttons or checkboxes, opening dialogs by button-click, etc.

In the spirit of MeetMath's didactical framework, the interactive visualizations serve mainly for motivation and acquisition. After introducing a new mathematical concept, we provide tools aiming at graphical visualiza-

tions of important properties that are to be investigated in the current unit. The user can interactively "play" with the tool, the on-line help gives instructions which phenomena can and should be observed. These interactive tools are always designed in such a fashion that in addition to pre-defined examples they allow to run user-defined examples as well as randomly generated examples. A symbolic computation system is indispensable as the engine behind these tools, because an instructive visualization often needs the computation of the problem's solution in the background. Symbolic computation methods can be applied for generating pre-computed symbolic solutions depending on example parameters, such that for visualization of a random example only the example parameters need to be instantiated in the symbolic solution.

During this phase, the students get an intuition about the new concept and in the best case they observe some of the intended properties during their experiments. Still staying in an "acquisition room" (see Section 2.1) we then formulate some conjectures in the *Theorema* language, which looks very much like standard mathematical formula language. Changing into a "strengthening room", we then ask the user to prove the conjecture using *Theorema*. Again, we provide a button interface for the prover call in order not to confuse the students with syntax details thereby distracting them from their main focus, the proof. The learning goal and, thus, the user interaction in this phase consists of choosing the appropriate knowledge base and observing its influence on the generated proof. Furthermore, students can investigate possible modifications in the formulation of the conjecture and/or parts of the knowledge in order to obtain a successful proof.

CreaComp consists of several thematic units that are intended for use in standard undergraduate courses for studies in mathematics and computer science as well as in mathematics courses for non-mathematical studies, e.g. business, marketing, social sciences, etc. Units are available for basic set theory, relations, functions, real-valued sequences and limits, continuous functions, fast computations using modular arithmetic, polynomial interpolation, Markov chains, cryptology, Gröbner bases, and other topics to be developed. We do not discuss content selection, i.e. the CreaComp environment does not define rules, *what* should be taught and what not and, in particular, what should be presented as white-box and what should be considered black-box. Rather, the above topics have been selected by the authors to be taught as white-boxes, and CreaComp provides a common frame *how* to present these in the particular computer environment.

In the remainder of this paper, we want to illustrate the structures

described above in a case study showing parts of the CreaComp unit on equivalence relations and set partitions.

3. The Case Study: Equivalence Relations and Set Partitions

Following the didactic principles taken from MeetMath, the typical flow of a CreaComp unit is to

- motivate the students by some real-world example,
- present new concepts by defining new objects or properties,
- let the students experiment with the new entities on concrete data,
- guide the students in their experiments such that possibly they are able to conjecture new properties,
- guide the students in rigorous proofs of their conjectures.

We try to illustrate the flavor of computer-support given in a CreaComp unit in the example chapter on "equivalence relations and set partitions".

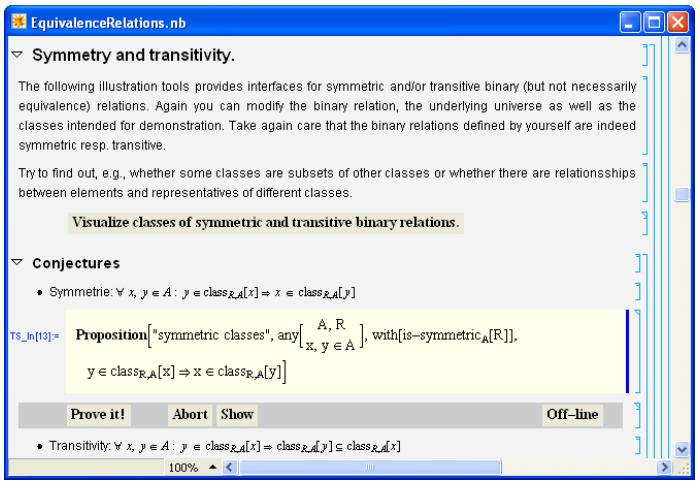


Fig. 1. A typical CreaComp educational unit.

Fig. 1 shows a screen-shot of a part of the notebook on equivalence relations containing the most important interactive interface elements. We see structured text containing inline mathematical formulae hierarchically grouped in nested cells intermixed with formal parts (e.g. definitions, theo-

rems, etc.) written in the *Theorema* language. *Theorema* blocks are written in *Mathematica* input cells and they differ in layout from the surrounding text blocks so that they can easily be recognized as active content. These cells must be evaluated in order for their content to be accessible in the *Mathematica*-kernel later. *Theorema* input can use most of the common mathematical notation[a], notably all sorts of quantifiers written in appealing two-dimensional form and, thus, *Theorema* input hardly differs from inlined mathematical formulae in the text.

Although *Mathematica* and *Theorema* provide palettes and keyboard shortcuts for inputting two-dimensional expressions, *Theorema* input is always prepared in advance and the users are usually not required to type *Theorema* formulae. Only occasionally, we leave parts of a formula blank and indicate with a "□"-placeholder that formula parts need to be inserted for the placeholder. CreaComp *interaction buttons* indicate the availability of an interactive experiment and they appear as gray boxes, the text above and on the button roughly explains the associated experiment. The unit shown in Fig. 1 contains an interaction button for visualizing classes of symmetric and transitive relations, which will be described in more detail in Section 3.1. Mathematical conjectures are formulated in *Theorema* language immediately followed by a CreaComp *prove panel* containing a prove-button, an abort-button, a show-button, and an off-line-button, see Section 3.2 for details.

3.1. *The Interface to Computer-Supported Experiments*

The interface to interactive experiments is always provided by so-called CreaComp interaction buttons. As an example, we describe the visualization tool behind the interaction button shown in Fig. 1. The notions "symmetry" and "transitivity" of a binary relation R on a universe A and the notion of a "class of an element x w.r.t. R and A" have been introduced earlier. At this point we want to investigate properties of classes when R is symmetric and/or transitive. When pressing the interaction button, a new window as shown in Fig. 2 appears on the screen. The window essentially contains 4 components:

[a]Some notation is supported already by standard *Mathematica*, like subscripts, fractions, summation/integration, and a bunch of special characters. Input syntax is configurable and *Theorema* uses this feature to support most of the common mathematical language. *Theorema* uses the braces $\{,\}$ for sets and anglebrackets $\langle,\rangle$ for tuples, which contrasts *Mathematica*, which uses braces for lists, i.e. tuples, and which does not have sets as distinguished objects.

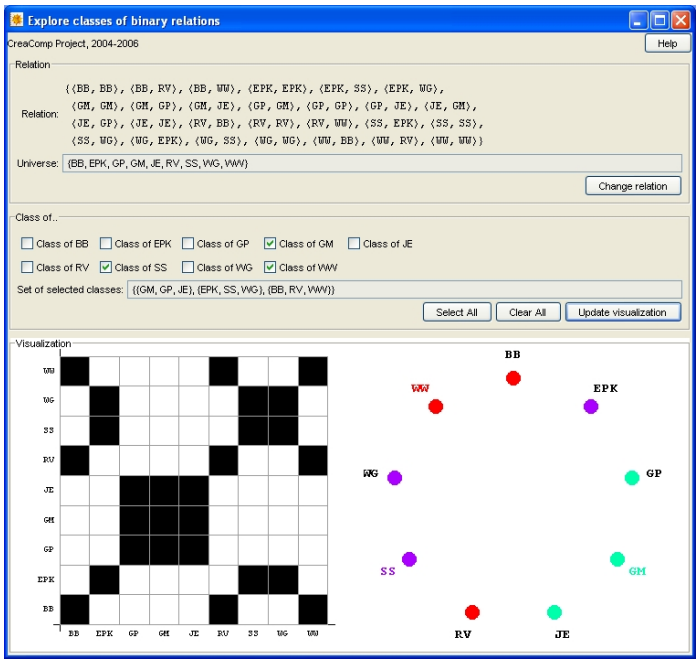


Fig. 2. Interactive visualization of classes.

(i) The top box displays the relation R as a set of pairs in *Theorema* notation and the universe set A. The relation R can be changed by either explicitly giving a new set of pairs or by having a random symmetric and/or transitive relation automatically generated by the system.

(ii) The middle box allows to select the classes to be visualized and it displays the respective classes as a set of sets in *Theorema* notation.

(iii) The bottom box shows graphical visualizations of the relation and the selected classes. The raster on the left corresponds to R's adjacency matrix and the arrangement of disks on the right indicates A's elements and each of the selected elements' class by color: each element is assigned a unique color shown by the color of its label and all elements in its class have their disk in the same color. Since intersecting classes are a key-feature to be discovered by this experiment, we display elements belonging to more than one class by a grey disk. The absence of grey disks in the visualization in Fig. 2 indicates disjoint classes in this example.

(iv) The help button in the top-right corner displays individual help for this

experiment by explaining which interactions can be made and which phenomena the student is expected to investigate.

Interactive visualizations are implemented using *Mathematica*'s GUIKit, which allows to build Java GUIs containing *Mathematica* data. In the example, the bottom graphics are re-calculated and re-drawn whenever there is user-interaction in one of the top boxes and this is the prototypical interaction pattern for CreaComp experiments: mathematical objects are visualized by *Mathematica* graphics that adapt to user-input given through intuitive interface elements such as check-boxes, radio-buttons, roll-down menus, etc., and special dialog windows that allow *Theorema* input that will be correctly parsed and processed before the respective graphics are re-generated. The on-line help explains the possible interactions and, as a side-effect, it is meant to lead the students' experiments into a "reasonable direction" so that they might conjecture "relevant knowledge". Students have the freedom in exploiting the interactive tools in arbitrary manner, but it is important to provide them some guidelines in what to try and what to observe, otherwise there is the danger that they get lost if they deviate from the intended track through the unit.

For all interactive tools, their design is learner centered, i.e. the developer of a unit needs to decide which visualizations are instructive at which place in a unit. For each case it needs to be decided whether available visualizations in *Mathematica* (including the numerous extension packages) can be re-used or whether specialized graphics need to be programmed. Finally, the interaction patterns need to be developed and implemented in the Java interface in such a fashion that the desired learning process is supported best. We observe that the *Mathematica* programming part is almost neglectable (regardless of availability of visualizations in packages) compared to the interface design and programming, having in mind unexperienced users that should intuitively follow intended paths as inspired by the interface layout and its behavior, respectively.

3.2. *The Interface to Automated Proving*

We discuss the CreaComp interface to *Theorema* provers using the example shown in Fig. 1. In plain *Theorema*, the command to generate the proof would be

Prove[Proposition["symmetric classes"], by → SetTheoryPCSProver,
using → {Definition["symmetry"], Definition["class"]}, ProverOptions →
{GRWTarget → {"goal","kb"}, UseCyclicRules → True},SearchDepth→50]

In practice we have often realized, that the call of the appropriate prover with the appropriate options turns out to be a real hurdle, not only for students. The problem is not syntax but a successful call requires detailed knowledge about available provers, their exact names, their options, and the influence of these options on the generation of the proof. Since the primary goal of the course is to learn about equivalence relations rather than learning how to use *Theorema* (or *Mathematica*), we decided to hide the concrete prover calls and instead provide a uniform interface to *Theorema*'s automated provers by a so-called CreaComp prove panel, see Fig. 1.

The prove panel is a highlighted part in the notebook directly following a proposition to be proven, and it consists of buttons controlling a *Theorema* prover. The prove-button on the left has a call to a *Theorema* prove method with appropriate parameters and options as shown above associated to its "button-pressed"-event. Every *Theorema* prover needs the knowledge base to be used in the proof as a parameter, thus, before actually starting the proof the user can compose the knowledge base in an interactive dialog[b]. Fig. 3 shows such a dialog window: it displays the formula to be proven and it lists all definitions, propositions, theorems, etc. available at this stage. The user simply selects by mouse-click and sends the knowledge base to the prover by pressing the "Prove"-button in the dialog's bottom-right corner. Pressing the "Hint"-button in the bottom-left corner selects just the appropriate portion of knowledge for a successful proof. The appropriate knowledge for a certain proof cannot be detected automatically, the developer of the unit needs to code this information within the prove panel so that the knowledge base composition tool can access it from there.

The abort-button in the prove panel aborts a running proof, the show-button shows the proof attempt, typically after having aborted the prover. The off-line-button on the right-margin of the prove panel allows to view a pre-generated successful proof. Both pre-generated and live-generated proofs appear in a separate window as shown in Fig. 4. The proof comes in human-readable format and explains each proof step in natural language.

3.3. *The Entire Unit*

After having explained the interaction possibilities that are spread all over the material whenever appropriate we can now browse through the unit "Equivalence Relations and Set Partitions", which introduces the students

[b]This feature will not be used in every proof. Sometimes the appropriate knowledge base will be hardcoded in the prove-button.

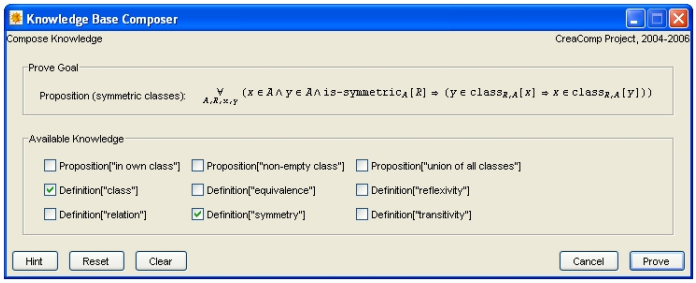


Fig. 3. Interactive knowledge base composition.

to the correspondence between equivalence relations and set partitions. First of all we introduce the mathematical objects to work with in this theory. The definitions are given in *Theorema* language and start with the definition of relations as a subset of some cartesian product. Since we want

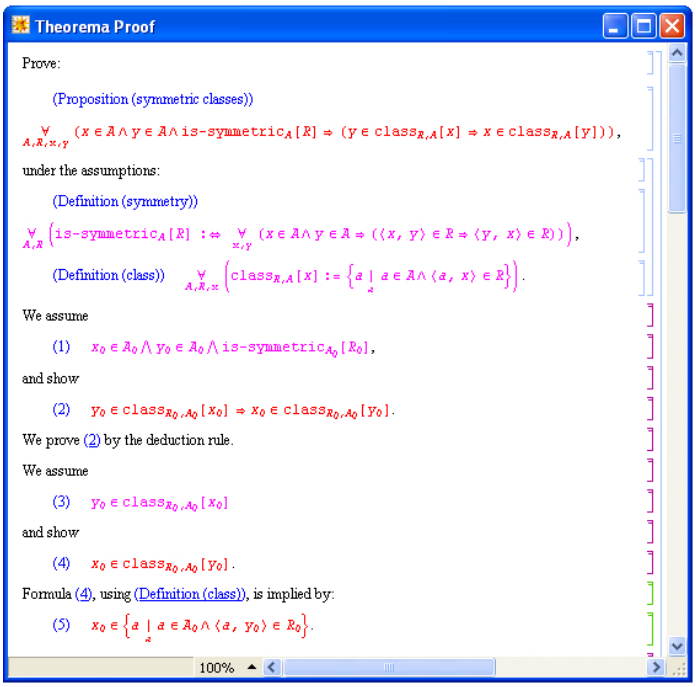


Fig. 4. *Theorema* proof.

to study classes and factor sets of equivalence relations, we restrict our theory to binary relations on some universe set. For other kinds of relations there are links to other CreaComp units, which focus on e.g. general relations or order relations. Furthermore, the first section of this unit defines the main properties interesting for equivalence relations, namely reflexivity, symmetry, and transitivity.

In order to develop some intuition on these properties of relations, we provide a visualization tool to illustrate these properties. The students can experiment with different relations, some are pre-defined, some are randomly generated or user-defined. This tool is similar to the one shown in Fig. 2 only that there is no mentioning of the concept of classes yet.

The next step is to introduce the concept of classes for binary relations. Traditionally, (equivalence) classes are defined for equivalence relations only. This is due to the nice properties of classes that can be proven in this setting. However, in the spirit of theory exploration, see e.g. [5], we find it interesting and natural to define *classes for arbitrary relations* and then study the dependence of properties of classes from properties of the underlying relation. In the *Theorema* language the definition of a class looks similar to common mathematical language, so the students can read and understand this definition easily without learning additional syntax:

Definition["class", any$[x, A, R]$,

$\mathsf{class}_{R,A}[x] := \{a \in A | \langle a, x\rangle \in R\}$]

Even for relations with only few properties, e.g. only reflexive, they can prove certain (weak) properties of classes; the stronger the properties of the relation the nicer the properties of the classes, until finally, towards the end of the unit, we find the classical results for equivalence classes. We think this evolution of the theory is an interesting aspect for students to experience.

Of course, conjectures proposed by the students or suggested in our material need not really be true. Some are false in general, but they may be valid in special cases. For example, after inspecting relations in the visualization tool some students might conjecture the following:

Proposition["in own class", any$[A, R]$,

$\underset{x \in A}{\forall}\ x \in \mathsf{class}_{R,A}[x]$]

After trying the proof they would realize that the prover fails to prove this proposition. In general, failure of a *Theorema* proof can have various reasons: the provided knowledge is insufficient or the proposition as stated is not provable, maybe not even true. This is just what we want to empha-

size in teaching mathematics, and conventional learning material does not provide room for this experience. Finally, failure can also be due to an inappropriate proving method or inappropriate parameters for the method, but we eliminate these sources by packing the call to the prover into the prove-button.

Theorema's possibility to inspect failing proof attempts comes very handy at this point, so students can investigate, which part of the proof led to failure. In the example above, they detect that the prover closes all branches successfully, only $\langle x_0, x_0\rangle \in R_0$ needs to be proven for arbitrary x_0 and R_0. Thorough inspection of the available knowledge at that stage shows that only $x_0 \in A_0$ is known and the student might learn that the proposition as stated *cannot be proven* unless more is known about R_0. In general, there are various possibilities how to modify the setup for the next proof attempt: the proof goal can be modified, assumptions can be changed, new assumptions can be added, or even a different proving method could be applied. We assist the student by writing a skeleton of a new formula into the notebook at this point, thereby fixing which kind of modification is sensible. In the above example, a side-condition to the proof goal is the way to go. In *Theorema* this can be done using the with[...] expression and we provide

Proposition["in own class", any[A, R], with[$\square$],

$\underset{x\in A}{\forall}\ x \in \mathsf{class}_{R,A}[x]$]

where the student is asked to substitute some condition for the "$\square$"-placeholder. Of course, in order to succeed with the above proof, R_0 must have the property that $\langle x, x\rangle \in R_0$ is true for all $x \in A_0$, i.e. R_0 must be reflexive on A_0. Hence, they must put the side-condition "is-reflexive$_A[R]$" and can then successfully prove the adapted proposition. This proof is not particularly difficult, still the formal rigor in which it is carried out is instructive for students.

The remaining content of this unit explains some properties of sets of sets. In particular, the students can learn about the factor set of a relation and set partitions and they can investigate properties that turn a set of sets into a partition. Moreover, the concept of an *induced relation* is introduced, namely

Definition["induced relation", any[A, S],

$\mathsf{induced\text{-}relation}_A[S] := \{\langle x, y\rangle \underset{x,y\in A}{|} \underset{M\in S}{\exists}\ x \in M \wedge y \in M\}$]

In each step the procedure is similar:

- *introduce* new objects or properties,
- *visualize* the properties in order to gain insights,
- *guess and propose* conjectures,
- *formalize* the conjectures precisely, and
- *prove* them automatically with *Theorema*.

The key observations are then:

- if R is an equivalence relation on A then factor-set$_A[R]$ is a partition of A and induced-relation$_A$[factor-set$_A[R]] = R$.
- if P is a partition of A then induced-relation$_A[P]$ is an equivalence relation and factor-set$_A$[induced-relation$_A[P]] = P$.

All these theorems including also all auxiliary lemmata necessary for compact proofs of the main statements are proved fully automatically within this learning unit. In order to demonstrate the readability of *Theorema* proofs, we show one proof in all details *as it is generated in the* Theorema *system*, compare its appearance also to Fig. 4.

Lemma["induced relation is transitive", any$[A, P]$, with[is-partition$_A[P]$], is-transitive$_A$[induced-relation$_A[P]]$]

In the knowledge base for this proof, we have the definitions of transitivity and induced relation and an auxiliary proposition proved earlier, namely

Proposition["intersecting classes are equal", any$[A, P]$, with[is-partition$_A[P]$], $\underset{X,Y\in P}{\forall} X \cap Y \neq \emptyset \Rightarrow X = Y$]

Proof. We assume

(1) is-partition$_{A_0}[P_0]$

and show

(2) is-transitive$_{A_0}$[induced-relation$_{A_0}[P_0]]$.

Formula (2), using (Definition (transitivity)), is implied by:

(3) $\underset{x,y,z\in A_0}{\forall} \langle x, y\rangle \in \text{induced-relation}_{A_0}[P_0] \wedge \langle y, z\rangle \in \text{induced-relation}_{A_0}[P_0] \Rightarrow \langle x, z\rangle \in \text{induced-relation}_{A_0}[P_0]$.

We assume

(4) $x_0 \in A_0 \wedge y_0 \in A_0 \wedge z_0 \in A_0 \wedge \langle x_0, y_0\rangle \in \text{induced-relation}_{A_0}[P_0] \wedge \langle y_0, z_0\rangle \in \text{induced-relation}_{A_0}[P_0]$

and show

(5) $\langle x_0, z_0\rangle \in \text{induced-relation}_{A_0}[P_0]$.

Formula (5), using (Definition (induced relation)), is implied by:

(8) $\langle x_0, z_0\rangle \in \{\langle x, y\rangle \underset{x,y\in A_0}{|} \underset{M\in P_0}{\exists} x \in M \wedge y \in M\}$.

In order to prove (8) we have to show

(9) $\underset{x,y\in A_0}{\exists} (\underset{M}{\exists} M \in P_0 \wedge x \in M \wedge y \in M) \wedge \langle x_0, z_0\rangle = \langle x, y\rangle$.

Since $x := x_0$ and $y := z_0$ solves the equational part of (9) it suffices to show

(10) $x_0 \in A_0 \wedge z_0 \in A_0 \wedge \underset{M}{\exists} M \in P_0 \wedge x_0 \in M \wedge z_0 \in M$.

Formula (10.1) is true because it is identical to (4.1)
Formula (10.2) is true because it is identical to (4.3)
Formula (4.4), by (Definition (induced relation)), implies:

(12) $\langle x_0, y_0\rangle \in \{\langle x, y\rangle \underset{x,y\in A_0}{|} \underset{M\in P_0}{\exists} x \in M \wedge y \in M\}$.

From (12) we know by definition of $\{T_x \underset{x}{|} P\}$ that we can choose an appropriate value such that

(13) $\underset{M}{\exists} M \in P_0 \wedge x1_0 \in M \wedge x2_0 \in M$,

(14) $\langle x_0, y_0\rangle = \langle x1_0, x2_0\rangle$.

Formula (14) simplifies to

(16) $x_0 = x1_0 \wedge y_0 = x2_0$.

By (13) we can take appropriate values such that:

(17) $M_0 \in P_0 \wedge x1_0 \in M_0 \wedge x2_0 \in M_0$.

Now, let $M := M_0$. Thus, for proving (10.3) it is sufficient to prove:

(21) $M_0 \in P_0 \wedge x_0 \in M_0 \wedge z_0 \in M_0$.

Formula (21.1) is true because it is identical to (17.1).
Formula (21.2), using (16.1), is implied by:

(22) $x1_0 \in M_0$.

Formula (22) is true because it is identical to (17.2).
Proof of (21.3) $z_0 \in M_0$: Formula (4.5), by (16.2), implies:

$\langle x2_0, z_0\rangle \in \text{induced-relation}_{A_0}[P_0]$

which, by (Definition (induced relation)), implies:

(23) $\langle x2_0, z_0\rangle \in \{\langle x, y\rangle \underset{x,y\in A_0}{|} \underset{M\in P_0}{\exists} x \in M \wedge y \in M\}$.

From (23) we know by definition of $\{T_x \underset{x}{|} P\}$ that we can choose an appropriate value such that

(24) $\underset{M}{\exists} M \in P_0 \wedge x3_0 \in M \wedge x4_0 \in M$,

(25) $\langle x2_0, z_0\rangle = \langle x3_0, x4_0\rangle$.

Formula (25) simplifies to

(27) $x2_0 = x3_0 \wedge z_0 = x4_0$.

By (24) we can take appropriate values such that:

(28) $M_1 \in P_0 \wedge x3_0 \in M_1 \wedge x4_0 \in M_1$.

Formula (21.3), using (27.2), is implied by:

(32) $x4_0 \in M_0$.

Formula (17.3), by (27.1), implies:

(33) $x3_0 \in M_0$.

From (28.2) together with (33) we know

(35) $x3_0 \in M_1 \cap M_0$.

From (35) we can infer

(36) $M_1 \cap M_0 \neq \emptyset$.

Formula (36), by (Proposition (intersecting classes are equal)), implies:

(37) $\underset{A,P}{\forall}$ is-partition$_A[P] \wedge M_0 \in P \wedge M_1 \in P \Rightarrow M_1 = M_0$.

Formula (1), by (37), implies:

(71) $M_0 \in P_0 \Rightarrow (M_1 \in P_0 \Rightarrow M_1 = M_0)$.

From (17.1) and (71) we obtain by modus ponens

(72) $M_1 \in P_0 \Rightarrow M_1 = M_0$.

From (28.1) and (72) we obtain by modus ponens

(73) $M_1 = M_0$.

Formula (32) is true because of (28.3) and (73). □

The proof as shown above will appear in a separate window and features interactive elements that cannot be rendered in the above "static reproduction": All formula references are active button elements, which will display the referenced formula in a separate window, proof goals and assumptions can easily be distinguished by color, see also Fig. 4, and the structure of the proof tree is reflected by the nested cell structure of the proof notebook, so that entire proof branches can be collapsed by a single mouse-click. In the configuration used for the above proof, *Theorema* does not display every single proof step it applies. In this example, for instance, it tacitly splits conjunctions in the goal and the knowledge base into its parts. This explains formula labels referring to formulae not actually present in the proof, e.g. formula (4.1) refers to the first conjunct in formula (4).

4. Conclusion and Future Work

CreaComp is work in progress. Therefore we do not have results on evaluation of the units in classroom yet. Further work will go into computer-supported assessment, which has already been implemented in the frame of MeetMath, see [11]. Assessment is heavily based on randomly generated test exercises based on example patterns, where the power of a symbolic computation system in the background is essential for both generating the exercises as well as checking correctness of user solutions. Although proving forms an essential part of our approach to teaching, we plan to assess neither the students' performance in proving nor their use of an automated theorem proving system. Rather, we test facts about mathematical concepts and we hope that proving enhances students' understanding of the mathematics involved. As a different branch, the use of *Theorema* provers for checking user answers can be investigated.

References

1. R.B. Andrews, C.E. Brown, F. Pfenning, M. Bishop, S. Issar, and H.Xi. ETPS: A System to Help Students Write Formal Proofs. *Journal of Automated Reasoning*, 32:75–92, 2004.
2. J. M. Borwein. The Experimental Mathematician: The Pleasure of Discovery and the Role of Proof. *International Journal of Computers for Mathematical Learning*, 10(2):75–108, May 2005.

3. B. Buchberger. Should Students Learn Integration Rules? *ACM SIGSAM Bulletin*, 24(1):10–17, January 1990.
4. B. Buchberger. Symbolic Computation: Computer Algebra and Logic. In F. Bader and K.U. Schulz, editors, *Frontiers of Combining Systems, Proceedings of FROCOS 1996 (1st International Workshop on Frontiers of Combining Systems), March 26-28, 1996, Munich*, volume 3 of *Applied Logic Series*, pages 193–220. Kluwer Academic Publisher, Dordrecht - Boston - London, The Netherlands, 1996.
5. B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger. Theorema: Towards Computer-Aided Mathematical Theory Exploration. *Journal of Applied Logic*, 4(4):470–504, 2006. ISSN 1570-8683.
6. B. Buchberger, C. Dupre, T. Jebelean, F. Kriftner, K. Nakagawa, D. Vasaru, and W. Windsteiger. The Theorema Project: A Progress Report. In M. Kerber and M. Kohlhase, editors, *Symbolic Computation and Automated Reasoning (Proceedings of CALCULEMUS 2000, Symposium on the Integration of Symbolic Computation and Mechanized Reasoning)*, pages 98–113. St. Andrews, Scotland, Copyright: A.K. Peters, Natick, Massachusetts, 6-7 August 2000.
7. B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuta, and D. Vasaru. A Survey of the Theorema Project. In W. Kuechlin, editor, *Proceedings of ISSAC'97 (International Symposium on Symbolic and Algebraic Computation, Maui, Hawaii, July 21-23, 1997), ACM Press*, pages 384–391, 1997. ISBN 0-89791-875-4.
8. P. Drijvers. Learning Mathematics in a Computer Algebra Environment: Obstacles are Opportunities. *Zentralblatt für Didaktik der Mathematik*, 34(5):221–228, 2002.
9. G. Hanna. Proof, Explanation and Exploration: An overview. *Educational Studies in Mathematics*, 44:5–23, 2000.
10. Florina Piroi and Temur Kutsia. The Theorema Environment for Interactive Proof Development. In G. Sutcliffe and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning. Proceedings of the 12th International Conference, LPAR'05*, volume 3835 of *Lecture Notes in Artificial Intelligence*, pages 261–275. Springer Verlag, 2005.
11. S. Saminger. MeetMATH — Visualizations and Animations in a Didactic Framework. In M. Borovcnik and H. Kautschitsch, editors, *Technology in Mathematics Teaching (Special groups and working groups). Proceedings of ICTMT 5, Klagenfurt (Austria)*, volume 26 of *Schriftenreihe Didaktik der Mathematik*, pages 217–222, Wien, 2002. öbv & hpt Verlagsgesellschaft.
12. R. Sommer and G. Nuckols. A Proof Environment for Teaching Mathematics. *Journal of Automated Reasoning*, 32:227–258, 2004.
13. R. Vajda. E-training of Formal Mathematics: Report on the CreaComp Project at the University of Linz, June 26 2006. Contributed talk at ACA 2006.
14. Stephen Wolfram. *The Mathematica Book*. Wolfram Media, Inc., 5th edition, 2003.

Stimulating Students' Creativity Through Computer-Supported Experiments and Automated Theorem Proving

Wolfgang Windsteiger
RISC Institute, JKU Linz, Austria

Introduction

We present an environment for learning and teaching mathematics that aims at inspiring the creative potential of students by enabling the learners to perform various kinds of interactive computer experiments during their learning process. Computer interactions are both of visual and purely formal mathematical nature, where the computer-algebra system *Mathematica* powers the visualization of mathematical concepts and the tools provided by the mathematical assistant system *Theorema*, which is also based on *Mathematica*, are used for the formal counterparts. We present case studies on equivalence relations and polynomial interpolation, in which we demonstrate the entire bandwidth of computer-support that we envision for modern learning environments for mathematics.

Motivation

Computers are used in mathematics education typically in order

- to perform computations that would be too time-consuming for a student to be done by pencil and paper, e.g. solve a 10x10-system of linear equations,
- to perform operations that a student is not able to at a certain level of education, e.g. solve a system of non-linear equations, or
- to visualize/animate mathematical objects, e.g. an animation of the sequence of Taylor polynomials depending on their degree.

However, such crucial mathematical skills like “the precise use of formal language”, “conjecturing mathematical properties”, “rigorously proving or disproving conjectures”, or “inventing mathematical algorithms” are often attributed to the human ingenuity or creativity and therefore their development is considered not supportable by computers.

We develop course material including *interactive experiments*, where in a first phase students are led to the discovery of mathematical properties or algorithms. In a second phase, they must precisely formulate their discoveries, and in a third phase they use the *Theorema* system to attempt an automated proof of the statement. The *Theorema* system gives a nicely structured human-readable proof in case of success, but even in case of failure it displays the incomplete proof. This allows students to reformulate their conjecture or to modify the knowledge base leading to an iterative process, in which they finally create a correct statement together with its correctness proof. For details on the *Theorema* system, we refer to [TMA].

We want to demonstrate these ideas in two case studies on equivalence relations and polynomial interpolation, respectively. The material is distributed in form of *Mathematica* notebooks, and *Mathematica* is the natural integrated user interface both for studying the material and for working with *Theorema*[1].

1 Of course, we assume that students have a modest understanding of the language of

Case Study: Equivalence Relations and Set Partitions

The material on equivalence relations and set partitions introduces binary relations on a universe and their properties such as reflexivity, symmetry, and transitivity. Next we introduce the concepts “class”, i.e. the set of all elements related to a given one, and “factor set”, i.e. the set of all classes. As soon as we present a new notion, we provide interactive visualization tools (so-called “widgets”), in which a student can experiment with the new entities in order to explore their properties.

Widgets are Java applications, which are linked to *Mathematica* through *JLink*, a communication protocol that allows to connect Java applications to a *Mathematica* kernel. Using JLink it is quite easy to setup graphical user interfaces (GUI) with typical interactive elements such as buttons, pull-down- or pop-up-menues, scrollbars, or text fields like in common modern user interfaces. In the background, however, these GUIs employ *Mathematica* for computations and graphics. In the case of relations and classes, several widgets are provided that use colors to visualize classes. The elements, whose class is to be displayed, the relation, and the universe can be chosen by the student and the widgets update with each user action. In some widgets, students may choose among predefined reflexive/symmetric/transitive relations, an example of a widget is shown in Figure 1.

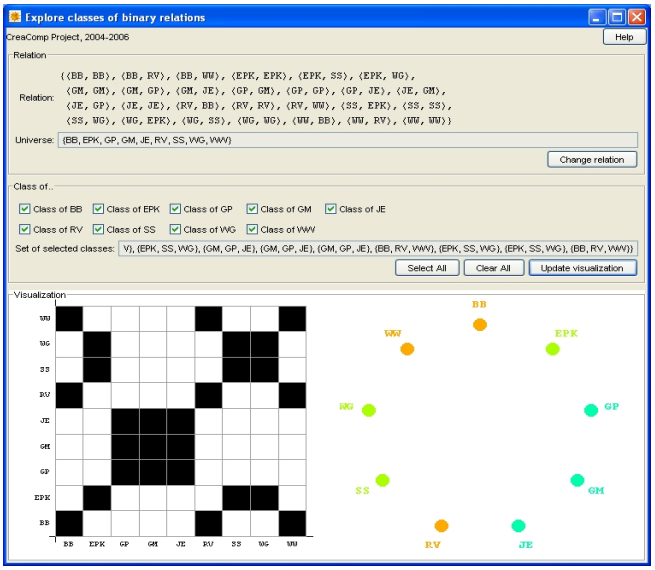


Figure 1: Example of a GUI for equivalence relations

The experiments with these tools establish an imagination about classes and their dependence on the relation's crucial properties. The classes' properties that are highlighted are non-emptyness and the “covering property”. After playing with these widgets a user could conjecture that for *equivalence relations* all classes are non-empty and each element is contained in exactly one of the classes, which is one of the key properties about equivalence classes. For a proof of this theorem, we proceed step by step using and proving several intermediate results on the way. In a first step, we concentrate on relexive relations only and the respective widget supports the idea that every element should be contained in it's own class. Whatever conjecture a student will make, we require them to state it *formally* using the *Theorema* language. In order to facilitate the input, *Mathematica* provides input palettes for special mathematical symbols, the *Theorema* system offers additional specialized palettes for mathematical formulae, and the learning material

mathematical formulae and they underwent training in basic handling of *Mathematica* and the *Theorema* system.

already contains templates that the students are only required to complete. For instance, after having interactively explored classes of reflexive relations, the students would be confronted with a Lemma-template of the form

Lemma[“in own class”, any[A,R], ...]

where “...” is a placeholder for any formula to be entered by the student. The *Theorema* system can now be employed to prove the lemma named “in own class”. In *Theorema*, the call of an automated prover needs the statement to be proven, a knowledge base, the prove-method, and sometimes additional optional parameters to be specified by the user. From the point of view of didactics, the most interesting aspect here is the specification of the knowledge base, which we emphasize by giving students the opportunity to experiment with different knowledge bases, while keeping all other parameters of the prove-call fixed. Therefore, after completing the the lemma-template above, the student can press a prove-button, which will first ask the user to compose a knowledge base before trying a completely automated proof. Again using GUI technology, arbitrary knowledge available in the current *Theorema* session can be adjoined to the knowledge base, and of course, the resulting proof heavily depends on the chosen knowledge. In case of an unsuccessful proof, the user can inspect the failing proof and then adjust the knowledge base, ask the system for a hint, or revisit the theorem. In the concrete example above, a student might complete the template to

Lemma[“in own class”, any[A, R], $\forall_x x \in class_{R,A}[x]$]

thereby forgetting the side-condition to force R to be reflexive on A. The proof of this statement cannot succeed, and the incomplete *Theorema* proof will display the reason for failure: for an arbitrary set A_0 , arbitrary $x_0 \in A_0$, and arbitrary relation R_0 on A_0 we need to show $\langle x_0, x_0 \rangle \in R_0$. The knowledge base, however, only contains the definitions of “class” and “reflexivity”, whereas nothing is known about R_0 . From this proof and an augmented Lemma-template, the student might get the insight that

Lemma[“in own class”, any[A, R], $reflexive_A[R] \Rightarrow \forall_x x \in class_{R,A}[x]$]

is a better way to formulate what has been observed in the interactive experiments earlier. The automated proof of this version of the Lemma goes through without problems. For details we refer to our article in [SCE].

Case Study: Polynomial Interpolation Algorithms

The case study on polynomial interpolation concentrates on strategies for algorithm development. We represent the interpolation polynomial P as a linear combination

$$P = \sum_{i=1}^{n} a_i B_i$$

of appropriate base polynomials B_i and show in a first attempt for an interpolation algorithm, that the basis {1, x , x^2 , ...} always leads to a system of linear equations for the coefficients in the linear combination, which in this case coincide with the polynomial coefficients. We then concentrate on grasping the idea of *Lagrangian interpolation*. We fix $a_i = y_i$, where y_i are the given values to be interpolated, and ask ourselves, whether the basis polynomials B_i can be adjusted such that P interpolates y_i on all given support points x_i . We provide a GUI again to adjust the basis polynomials, where a student can

- enter support points x_i and the respective values y_i and
- adjust the values of $B_j(x_i)$ for each combination of *i,j*.

The main window of the GUI then displays all interpolation points (x_i, y_i), all basis polynomials B_i, and the interpolation polynomial *P* as shown in Figure 2.

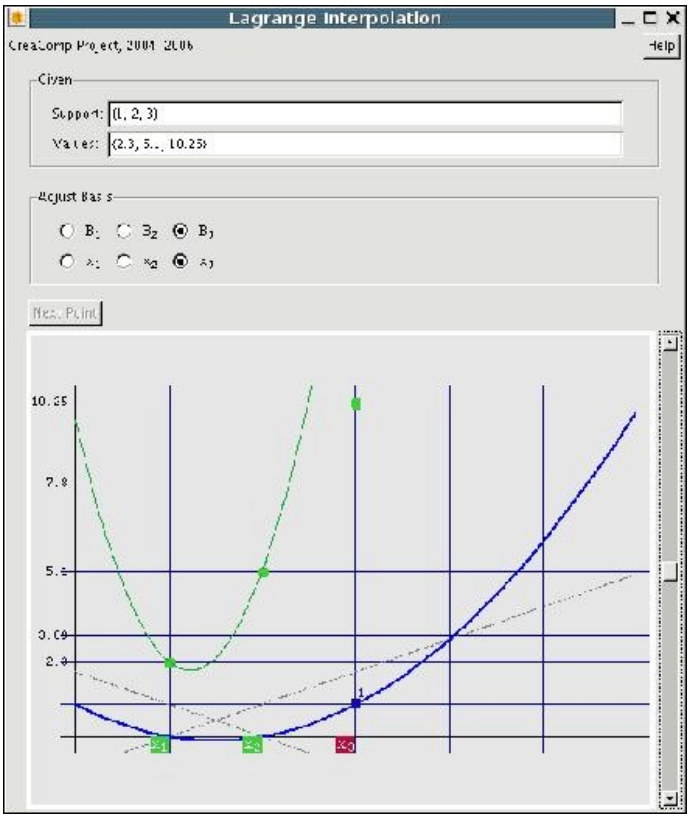


Figure 2: GUI for adjusting Lagrange basis polynomials

Support points x_i are marked green, if $P(x_i)=y_i$, otherwise they are red. The GUI forces the student to iterate *j=1,...,n*, i.e. it starts with *j=1* and there is only one basis polynomial B_1, whose value needs to be chosen at x_1. Once all points x_i are green for *i=1,...,j* the student is allowed to proceed the next interpolation point, i.e. increase *j* by 1. All basis polynomials $B_1,\ldots,B_j$ are left unchanged from the previous step, there is a new basis polynomial B_{j+1}, and the student must (re-)adjust $B_k(x_i)$ for all *i,k=1,...j+1* using the radio buttons on top and the scrollbar available to the right. The degrees of all basis polynomial increases by 1 automatically, and it is not too difficult to conjecture the crucial property of the *Lagrange basis polynomials*, namely $B_i(x_i)=1$ and $B_j(x_i)=0$ for all $i \neq j$. In a similar experiment, students can then explore the well-know product-formula for the basis polynomials.

In a second part, we let students explore the Neville-algorithm for interpolation, which can be formulated as a recursion that computes a degree *n-1* interpolation polynomial at *n* support points from two degree *n-2* polynomials interpolating at the first and the last *n-1* support points, respectively. Similar to the style shown in the equivalence relation case study, students can prove the main theorem on Neville's algorithm, and a GUI allows to

inspect the recursive behaviour of the algorithm. It shows that straight-forward recursive implementation would lead to a tremendously inefficient procedure and leads students to the idea of organizing the algorithm in an iteration according to the “Neville scheme”.

Conclusion

We have presented interactive course material based on *Mathematica* and *Theorema*. Its main feature is the *stimulation of mathematical creativity* through *interactive experiments* and the *integration of automated proving into mathematics teaching*. This approach puts the students' own activity into the center of attention and it encourages them to *explore mathematics* by themselves as opposed to just *present facts* as is often practiced in traditional maths teaching. The material can accompany conventional teaching in the classroom, but it can as well be used for distance teaching, home study or similar learning scenarios.

Acknowledgements

The work presented in this paper is joint work with Susanne Saminger and Günther Mayrhofer in the frame of the project “CREACOMP” directed by Prof. Bruno Buchberger, Prof. Erich Peter Klement, and Prof. Günter Pilz at JKU Linz.

References

[TMA] B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, W. Windsteiger. *Theorema: Towards Computer-Aided Mathematical Theory Exploration*. Journal of Applied Logic **4**(4), pp. 470-504. 2006. ISSN 1570-8683.

[SCE] G. Mayrhofer, S. Saminger, W. Windsteiger. *CreaComp: Experimental Formal Mathematics for the Classroom*. In: Symbolic Computation and Education, Shangzhi Li, Dongming Wang, and Jing-Zhong Zhang (ed.), pp. 94-114. 2007. World Scientific Publishing Co., Singapore, New Jersey, ISBN 978-981-277-599-3.

About the Author

Dipl.-Ing. Dr. Wolfgang Windsteiger

RISC Institute

Johannes Kepler University Linz

Schloss Hagenberg,

A-4232 Austria.

email: Wolfgang.Windsteiger@risc.uni-linz.ac.at

Phone: +43 732 2468 9960

Fax: +43 732 2468 29960

2 Related Publications

This section adds some relevant related publications, which are not integral part of this thesis.

[Windsteiger, 2001], [Windsteiger, 2002a], and [Windsteiger, 2002b] are all seriously refereed papers on the set theory prover in Theorema 1. They are not part of the main thesis since they do not add substantial innovations compared to what is contained in Paper 1 of the thesis except for maybe a few more examples of automatically generated proofs.

[Jebelean et al., 2009] is another survey article about the Theorema system similar to Paper 2 in the thesis. The author contributed a description of the set theory prover, the Theorema language, and the integration of computation and reasoning. These topics are covered by the Papers 1, 2, and 3 in the thesis. For this reason and also due to the weak refereeing process we do not include this paper into the main part of the thesis.

The article [Windsteiger, 2012] also contains a presentation of the new features in Theorema 2 but did not undergo as strict refereeing as Paper 5 in the main part of the thesis. We would like to point to a second paper on Theorema 2 because the novelties in this system exceed by far what is presented in this thesis.

The final project report of the CreaComp project [Buchberger et al., 2006] can be consulted as supplementary material underpinning the claims of Papers 10 through 12, since it contains all learning units that have been developed in the frame of this project.

[Windsteiger, 1996] and [Windsteiger, 1999] are fully interactive electronic lecture notes about probability calculus and statistics. They are of course not refereed in any sense, but they are early forerunners of the executable predicate logic language of Theorema. In fact, the lecture notes are based on a first prototype implementation of the Theorema language in Mathematica 3. The main concepts implemented later in Theorema had their testbed in this material, and it is an early example of using a mathematical assistant system in the context of mathematics education. Moreover, since many of the computations in probability are based on finite set computation, the work on this material was the initial motivation to add reasoning capabilities for set theory, which then led to the implementation of the set theory prover in Theorema 1 described in Paper 1 of this thesis.

Finally, [Windsteiger, 2003] is a strongly refereed book contribution, whose topic and content do not fit into the frame of this thesis. However, it proves the author's expertise in symbolic computation.

References

B. Buchberger, E. Klement, G. Pilz, S. Saminger, and W. Windsteiger. CreaComp: e-Schulung von Kreativität und Problemlösekompetenz. RISC Report Series 06-09, Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Schloss Hagenberg, 4232 Hagenberg, Austria, 2006.

T. Jebelean, B. Buchberger, T. Kutsia, N. Popov, W. Schreiner, and W. Windsteiger. Automated Reasoning. In B. Buchberger, M. Affenzeller, A. Ferscha,

M. Haller, T. Jebelean, E. Klement, P. Paule, G. Pomberger, W. Schreiner, R. Stubenrauch, R. Wagner, G. Weiß, and W. Windsteiger, editors, *Hagenberg Research*, pages 63–101. Springer Dordrecht Heidelberg London New York, 2009. ISBN 978-3-642-02126-8. URL `http://www.springer.com/computer/programming/book/978-3-642-02126-8`.

W. Windsteiger. Algorithmische Mathematik VI. Lecture notes for the mathematics course in the sixth semester at the Fachhochschule for Software Engineering in Hagenberg, Austria, in german. Available at `www.risc.jku.at/people/wwindste/Teaching/AlmaVI/Skriptum/`, 1996.

W. Windsteiger. Algorithmische Mathematik IV. Lecture notes for the mathematics course in the fourth semester at the Fachhochschule for Software Engineering in Hagenberg, Austria, in german. Available at `www.risc.jku.at/people/wwindste/Teaching/AlmaVI/Skriptum/`, 1999.

W. Windsteiger. A Set Theory Prover in *Theorema*. In R. Moreno-Diaz, B. Buchberger, and J. Freire, editors, *Computer Aided Systems Theory*, number 2178 in LNCS, pages 525–539. Springer, 2001. Proceedings of EUROCAST 2001 (8th International Conference on Computer Aided Systems Theory – Formal Methods and Tools for Computer Science), ISBN 3-540-42959.

W. Windsteiger. An Automated Prover for Set Theory in Theorema. In O. Caprotti and V. Sorge, editors, *Calculemus 2002, 10th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning: Work in Progress Papers*, pages 56–67, Marseille, France, June 2002a. ISBN 1427-4447 (ISSN). Seki-Report Series Nr. SR-02-04, Universität des Saarlandes.

W. Windsteiger. An Automated Prover for Zermelo-Fraenkel Set Theory in Theorema. In K. Nakagawa, editor, *Logic, Mathematics and Computer Science: Interactions (LMCS 2002)*, pages 266–280, RISC, Schloss Hagenberg, Austria, October 2002b. ISBN 3-902276-03-7. Symposium in Honor of Bruno Buchberger's 60th Birthday, RISC-report Series Nr. 02-60.

W. Windsteiger. Mathematica. In J. Grabmeier, E. Kaltofen, and V. Weispfenning, editors, *Computer Algebra Handbook: Foundations, Applications, Systems*, pages 314–320. Springer, 2003. ISBN 3-540-65466-6.

W. Windsteiger. Theorema 2.0: A Graphical User Interface for a Mathematical Assistant System. In J. Davenport, J. Jeuring, C. Lange, and P. Libbrecht, editors, *24th OpenMath Workshop, 7th Workshop on Mathematical User Interfaces (MathUI), and Intelligent Computer Mathematics Work in Progress*, number 921 in CEUR Workshop Proceedings, pages 73–81, Aachen, 2012. ISBN 1613-0073 (ISSN). URL `http://ceur-ws.org/Vol-921/`.

CHAPTER III

CURRICULUM VITAE

Family

Date of birth: December 31, 1967 in Linz, Austria.

Father: Ing. Hans Windsteiger, departmental head in a tiling company, retired since 1987, died April 9, 1999.

Mother: Cäcilia Windsteiger, housewife, died May 28, 2008.

Brothers and sisters: 1 brother, 1 sister.

Marital Status: Married to Mag. Cornelia Altreiter-Windsteiger.

Education

Primary school

1974–1978: Volkschule 42 in Linz/St. Magdalena.

High school

1978–1986: Bundesrealgymnasium Linz-Auhof, Aubrunnerweg 4. Final examination on June 10, 1986 with excellent success.

University

- 1986–1992: Study of "*Technical Mathematics*", branch "*Data- and Informationprocessing*", at the University of Linz, Austria.

  - First diploma exam (1. Diplomprüfung) on February 7, 1989.
  - Final exam (2. Diplomprüfung) on March 26, 1992 with excellent success.
  - Diploma thesis: "*Gröbner Bases: A Characterization by Syzygy Completeness and an Implementation*" at the RISC–Institute supervised by Prof. Bruno Buchberger, see [Windsteiger, 1992].
  - Graduation (Dipl.-Ing.) on May 8, 1992.

– 1992–2001: PhD study at the RISC–Institute supervised by Prof. Bruno Buchberger.

  - PhD thesis: "*A Set Theory Prover in Theorema: Implementation and Practical Applications*" at the RISC–Institute supervised by Prof. Bruno Buchberger, see [Windsteiger, 2001d].
  - Final exam (Rigorosum) on June 13, 2001 with excellent success.

Awards

– Erwin Wenzl Prize (category: university) for best PhD thesis in the year 2001.

Civil Service

1. Oktober 1996 – 31. August 1997: Red Cross Upper Austria.

Positions

– Since 1. November 2000: Assistant professor (Universitätsassistent) at the University of Linz.

– January 1993 – September 2000: Research assistant (Forschungsassistent) at RISC Linz.

– January 1990 – December 1992: Project co-worker in the project "Gröbner Bases" directed by Bruno Buchberger at RISC Linz.

– Since 1993: Lecturer at the University of Linz.

– Since 1993: External lecturer at the University of Applied Sciences (Fachhochschule) Hagenberg.

Research

Gröbner Bases

Projects

– January 1990 – December 1993: Research project "*Gröbner Bases*" lead by Prof. Bruno Buchberger sponsored by "Österreichischer Fond zur Förderung

der wissenschaftlichen Forschung" of the austrian government. Implementation of (parts of) the Gröbner bases algorithm in several systems and languages (Portable Common Lisp (PCL), *Mathematica*, C, SACLIB-C).

– January 1995 – September 1996: Numerical methods for Gröbner basis computation in the frame of a cooperation with Fujitsu Laboratories Ltd. and the Technical University of Vienna (Inst. for Numeric and Applied Mathematics, Prof. Stetter).

Publications & Talks

– Thesis: [Windsteiger, 1992].

– Technical Reports: [Windsteiger, 1990a], [Windsteiger, 1990b], [Windsteiger and Buchberger, 1993], [Windsteiger, 1993a], [Windsteiger, 1993b].

– Invited Talks at Seminars: [Windsteiger, 2006e].

– Talks at Conferences & Seminars: [Windsteiger, 1994], [Windsteiger, 1995a], [Windsteiger, 2006e], [Windsteiger, 2005a].

Theorem Proving

Projects

– September 1994 – December 1994: Research project for implementing a "*New Math Software System*" under the direction of Prof. Bruno Buchberger at RISC.

– since September 1997: Research project "*Theorema*" for developing a system for automated natural-style-proving in various mathematical domains based on *Mathematica* under the direction of Prof. Bruno Buchberger at RISC.

– February 2004 – December 2006: Research project "CreaComp" in combining the software systems *Theorema* and MeetMath for computer-supported teaching of Mathematics at university level under the direction of Prof. B. Buchberger, Prof. E.P. Klement, and Prof. G. Pilz at JKU Linz, see [Buchberger et al., 2006b].

Publications & Talks

– Articles: [Windsteiger, 1999a], [Windsteiger, 2001f], [Windsteiger, 2001b], [Windsteiger, 2002d], [Windsteiger, 2002e], [Windsteiger, 2003e], [Buchberger et al., 2006a], [Windsteiger et al., 2006], [Windsteiger, 2006a], [Clarke et al., 2006], [Mayrhofer et al., 2007a], [Mayrhofer et al., 2007b], [Windsteiger, 2008a], [Jebelean et al., 2009], [Kerber et al., 2011], [Windsteiger, 2012b], [Windsteiger, 2012c], [Kerber et al., 2013], [Lange et al., 2013].

– Thesis: [Windsteiger, 2001d].

- Technical Reports: [Windsteiger, 2001e], [Windsteiger, 2001a].
- Invited Talks at Conferences & Seminars: [Windsteiger, 2005b], [Windsteiger, 2005c], [Windsteiger, 2005d], [Windsteiger, 2006b], [Windsteiger, 2006c], [Windsteiger, 2009], [Windsteiger, 2013b].
- Talks at Conferences & Seminars: [Windsteiger, 1995b], [Buchberger and Windsteiger, 1998], [Windsteiger, 1999b], [Windsteiger, 2001c], [Windsteiger, 2002a], [Windsteiger, 2002b], [Windsteiger, 2003b], [Windsteiger, 2002c], [Windsteiger, 2005e], [Windsteiger, 2003a], [Windsteiger, 2003c], [Windsteiger, 2005b], [Windsteiger, 2006d], [Windsteiger, 2006f], [Windsteiger, 2007a], [Windsteiger, 2007b], [Windsteiger, 2008b], [Windsteiger, 2010], [Kerber and Windsteiger, 2011], [Windsteiger, 2011], [Windsteiger, 2012a], [Windsteiger, 2012d], [Windsteiger, 2013a].

Algorithms

- Book project "Algorithmische Methoden" in cooperation with Dr. Philipp Kügler (JKU Linz) in the new series "Mathematik kompakt" by Birkhäuser/Springer (Basel), see [Kügler and Windsteiger, 2008, 2012].

Research Visits

- August 1993 – September 1993: Visiting researcher at the *Department of Electronics and Electric Engineering* of the University of Pretoria, South Africa.
- April 25 – April 29, 2005: Visiting researcher in the frame of an ERASMUS exchange program at the *Department of Algebra (Prof. Jiri Tuma)* of the Charles University, Prag, Czech Republic.
- January 16 – April 30, 2006: Visiting researcher in the Analytica project at the *Department of Computer Science (Prof. Edmund M. Clarke & Prof. Klaus Sutner)* at Carnegie Mellon University, Pittsburgh, USA.

Research Organization

- Co-editor of "Hagenberg Research", see [Buchberger et al., 2009].
- General chair, program committee co-chair, and local chair for "Calculemus'2007", see [Kauers et al., 2007a], [Kauers et al., 2007b].
- General chair and local chair for "MKM'2007".
- Program chair "CICM'2013/Calculemus Track".
- General co-chair for "GeoGebra'2011".
- General co-chair for "GeoGebra'2009".
- General co-chair for "CADGME'2009".
- General co-chair for "ACA'2008".

- Workshop chair for “CIAO’2010”.
- Workshop co-chair of the workshop “Mathematical Theory Exploration” in the frame of *The 4th International Congress on Mathematical Software* (ICMS’2014, a satellite event of the International Congress of Mathematicians (ICM’2014), Seoul, Korea.
- Workshop co-chair of the workshop “Computer-Supported Mathematical Theory Development” in the frame of IJCAR’2004, Cork, Ireland (see [Benzmüller and Windsteiger, 2004]).
- Local chair and publicity chair LMCS’02, Hagenberg, Austria.
- Co-Organizer of mini-symposium “Proving in Mathematics Education at University and at School” (CSASC’2013), Koper, Slovenia.
- Co-organizer of the “CAL”-workshop (Computer Algebra and Automated Theorem Proving) at EUROCAL’2001, Gran Canaria, Spain.
- Member of program committee for “Calculemus” (2001–2003, 2007, 2008, 2010).
- Member of program committee for “CICM” (2012).
- Member of program committee for “AISC” (2004, 2008, 2010, 2014).
- Member of program committee for “UITP’2014”.
- Member of program committee for “SETS’2014”.
- Member of program committee for “Theorem-Prover based Systems for Education (eduTPS)” at CADGME’12.
- Member of program committee for “FroCoS’2011”.
- Member of program committee for “Automatheo’2010”.
- Member of program committee for “MIPS’2010”.
- Member of program committee for “CADGME’2009”.
- Member of program committee for “PLMMS’2007”.
- Member of program committee for “IJCAR’2004”.
- Member of program committee for “Electronic Journal of Mathematics & Technology (eJMT)”, Special Issue February 2013: Theorem-Prover based Systems for Education.
- Calculemus trustee (2000–2003, 2006–2009, 2012–2015).
- Member of the organizing committee “FPSAC’2009”.

Teaching

Lectures

- "*Algorithmic Methods 1*" obligatory in the first semester for the study of Technical Mathematics at the University of Linz: since WS01.
- "*Programming in Mathematica*" at the University of Linz: since SS96.
- "*Predicate Logic as a Working Language*" obligatory in the second semester for the study of Technical Mathematics at the University of Linz: SS03–SS13.
- "*Computer-based Working Environments*" at the JKU Linz: WS08–WS12.
- "*Algebraic and Discrete Methods in Biology*" at the JKU Linz: SS07–SS09.
- "*Logical and Formal Foundations of Computer Science*" at the Fachhochschule Hagenberg (branch Software Engineering): since WS08.
- "*Mathematics 1 — Algebra*" at the Fachhochschule Hagenberg (branch Software Engineering): since SS09.
- "*Formal Problem Solving*" at the Fachhochschule Hagenberg (branch Mobile Computing): WS08.
- "*Präsentationstechnik*" obligatory in the third semester for the study of Technical Mathematics at the University of Linz: WS03.
- "*Mathematics I*" at the Fachhochschule Hagenberg (branch Computer-based Learning): WS05.
- "*Mathematics II*" at the Fachhochschule Hagenberg (branch Computer-based Learning): SS06.
- "*Algorithmic Mathematics III*" at the Fachhochschule for Software Engineering in Hagenberg: WS94, WS95.
- "*Algorithmic Mathematics IV*" at the Fachhochschule for Software Engineering in Hagenberg: SS98, SS99.
- "*Algorithmic Mathematics VI*" at the Fachhochschule for Software Engineering in Hagenberg: SS96, SS97, SS98.
- "*Algorithmic Mathematics VIII*" at the Fachhochschule for Software Engineering in Hagenberg: SS98.
- "*Applied Mathematics*" at the "Ergänzungslehrgang" in the Fachhochschule for Software Engineering in Hagenberg: WS95 – SS99.

Exercises

- "*Algebra for Computer Scientists*" at the University of Linz: since SS14.
- "*Logic for Computer Scientists*" at the University of Linz: since WS13.
- "*Diskrete Strukturen*" at the University of Linz: since WS13.
- "*Mathematische Grundlagen 1*" at the University of Linz: WS07–WS12.
- "*Formale Grundlagen der Informatik I*" at the University of Linz: WS99 and WS04 – WS06.
- "*Mathematik I (Analysis)*" at the University of Linz: SS00 – SS01.
- "*Mathematik für Informatiker III*" at the University of Linz: WS93 – WS99.
- "*Logical and Formal Foundations of Computer Science*" at the Fachhochschule Hagenberg (branch Software Engineering): since WS08.
- "*Mathematics 1 — Algebra*" at the Fachhochschule Hagenberg (branch Software Engineering): since SS09.
- "*Algorithmic Mathematics I*" at the Fachhochschule for Software Engineering in Hagenberg: WS93 – WS02.
- "*Algorithmic Mathematics II*" at the Fachhochschule for Software Engineering in Hagenberg: SS94 and SS00 – SS03.
- "*Algorithmic Mathematics III*" at the Fachhochschule for Software Engineering in Hagenberg: WS94 – WS95.
- "*Algorithmic Mathematics IV*" at the Fachhochschule for Software Engineering in Hagenberg: SS95, SS99.
- "*Algorithmic Mathematics V*" at the Fachhochschule for Software Engineering in Hagenberg: WS98.
- "*Algorithmic Mathematics VI*" at the Fachhochschule for Software Engineering in Hagenberg: SS96.

Seminars

- Seminar in the frame of "Schwerpunktfach Mathematik", Europagymnasium Auhof, December 15, 2005. [Windsteiger, 2005f].
- "*Scientific Writing & Presentation*" at the Fachhochschule Hagenberg (branches Media-technology & -design, Hardware/Software Systems Engineering): SS05.
- "*Die Sprache der Mathematik*": Seminar at the "BORG für Kommunikation", Hagenberg, March 2, 2005.
- Course for mathematics teachers given in the frame of the "Tag der Mathematik 2001" at the University of Linz, November 23, 2001. [Windsteiger, 2001a].

- Project leader in "Projektwoche Angewandte Mathematik" for gifted high school students organized by Stiftung Talente in cooperation with the University of Linz: February 14-18, 2004, February 13-17, 2005.
- Organizer and teacher of seminars for *Mathematica*: October 1994 to December 2008.

Supervision of Master's Theses

- Laura Giuri, [Giuri, 2010].
- Dietmar Kerbl, [Kerbl, 2010].
- Marek Sacha, [Sacha, 2011].
- Shereen El Bedewy, [Bedewy, 2013].

Organization

- Director of the "Ergänzungslehrgang" in the Fachhochschule for Software Engineering in Hagenberg: WS95 – SS99.

Further Project Experience

- February 2002 – June 2002: Industrial project in optimization of transport and scheduling of production (Salinen Austria).
- August 1999 – August 2000: Co-worker in a research project "IMMENSE" between the University of Linz an the Austrian Ministry for Education, Science, and Culture for developing electronic courseware for teaching mathematics at universities based on *Mathematica*. Project directed by Prof. E.P Klement and Prof. G. Pilz at JKU Linz. Leader of technical team, responsible for overall system design and software technological concept, major developer of user interface components.
- January 1994 – August 1994: Industrial project in automatization of window production (Actual).

University Administration

- June 2001 – July 2005: Member of the faculty for science and engineering (TNF) at the University of Linz.
- Member of the curriculum commission "Doctorate in Technical Sciences" at JKU.
- Member of the curriculum commission "Technical Mathematics" at JKU.
- Support at SIM (general study information exposition at JKU).
- Support at "Traumberuf Technik" (information exposition for technical studies at JKU).

– Teaching at "Projektwoche Angewandte Mathematik" for talented pupils in Upper Austria.

Miscellaneous

– April 1991 – December 2008: Product manager and seminar leader for *Mathematica* and MAPLE (until December 1993) in the company UNI SOFTWARE PLUS.

– Contributed article in the *Handbook of Computer Algebra*, see Windsteiger [2003d].

– Design and implementation of the "RISC Activity Database" for maintaining records of publications and other scientific activities of RISC members[1].

– Organization of "*20 Years RISC*" celebration June 2008[2].

– Organization and moderation of the celebration in honor of Bruno Buchberger's 70th birthday, February 14, 2013.

– Moderation of the opening ceremony for the new RISC building, June 26, 2013[3].

Complete List of Publications

S. E. Bedewy. Gesture-Based Browsing of Mathematics. Master's thesis, Internationaler Universitätslehrgang Informatics: Engineering and Management (ISI), Johannes Kepler University Linz, 2013.

C. Benzmüller and W. Windsteiger, editors. *Computer-Supported Mathematical Theory Development*, University College Cork, Ireland, July 2004. ISBN 3-902276-04-5. URL `http://www.risc.uni-linz.ac.at/about/conferences/IJCAR-WS7/`. Workshop on Computer-Supported Mathematical Theory Development in the frame of IJCAR'04.

B. Buchberger and W. Windsteiger. The *Theorema* Language: Implementing Object- and Meta-Level Usage of Symbols. *Calculemus Workshop*, 1998.

B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger. Theorema: Towards Computer-Aided Mathematical Theory Exploration. *Journal of Applied Logic*, 4(4):470–504, 2006a. doi: http://dx.doi.org/10.1016/j.jal.2005.10.006.

B. Buchberger, E. Klement, G. Pilz, S. Saminger, and W. Windsteiger. CreaComp: e-Schulung von Kreativität und Problemlösekompetenz. RISC Report Series 06-09, Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Schloss Hagenberg, 4232 Hagenberg, Austria, 2006b.

[1] `http://www.risc.jku.at/publications/`
[2] `http://www.risc.jku.at/conferences/20YRISC2008/`
[3] `http://www.risc.jku.at/about/extension/`

B. Buchberger, M. Affenzeller, A. Ferscha, M. Haller, T. Jebelean, E. Klement, P. Paule, G. Pomberger, W. Schreiner, R. Stubenrauch, R. Wagner, G. Weiß, and W. Windsteiger, editors. *Hagenberg Research*, 2009. Springer Dordrecht Heidelberg London New York. ISBN 978-3-642-02126-8. URL `http://www.springer.com/computer/swe/book/978-3-642-02126-8`.

E. M. Clarke, A. S. Gavlovski, K. Sutner, and W. Windsteiger. Analytica V: Towards the Mordell-Weil Theorem. In A. Bigatti and S. Ranise, editors, *Proceedings of Calculemus'06*, pages 35–50, 2006.

L. Giuri. Automated Contract Generation and Document Management in the Context of Economic Promotional Business. Master's thesis, Internationaler Universitätslehrgang Informatics: Engineering and Management (ISI), Johannes Kepler University Linz, 2010.

T. Jebelean, B. Buchberger, T. Kutsia, N. Popov, W. Schreiner, and W. Windsteiger. Automated Reasoning. In B. Buchberger, M. Affenzeller, A. Ferscha, M. Haller, T. Jebelean, E. Klement, P. Paule, G. Pomberger, W. Schreiner, R. Stubenrauch, R. Wagner, G. Weiß, and W. Windsteiger, editors, *Hagenberg Research*, pages 63–101. Springer Dordrecht Heidelberg London New York, 2009. ISBN 978-3-642-02126-8. URL `http://www.springer.com/computer/programming/book/978-3-642-02126-8`.

M. Kauers, M. Kerber, R. Miner, and W. Windsteiger, editors. *Towards Mechanized Mathematical Assistants*, volume 4573 of *Lecture Notes in Computer Science*, Heidelberg, 2007a. Springer. ISBN 0302-9743 (ISSN). URL `http://www.springeronline.com/978-3-540-73083-5`. Proceedings of Calculemus 2007 and MKM 2007.

M. Kauers, M. Kerber, R. Miner, and W. Windsteiger. Calculemus/MKM 2007 – Work in Progress. RISC Report Series 07-06, Research Institute for Symbolic Computation (RISC), Johannes Kepler University of Linz, Schloss Hagenberg, 4232 Hagenberg, Austria, 2007b.

M. Kerber and W. Windsteiger. Using Theorema in the Formalization of Theoretical Economics, July 22 2011. Contributed talk at CICM 2011.

M. Kerber, C. Rowat, and W. Windsteiger. Using Theorema in the Formalization of Theoretical Economics. In J. H. Davenport, W. M. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, volume 6824 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 58–73. Springer, 2011. ISBN 0302-9743 (ISSN). URL `http://dx.doi.org/10.1007/978-3-642-22673-1_5`.

M. Kerber, C. Lange, C. Rowat, and W. Windsteiger. Developing an Auction Theory Toolbox. In M. Kerber, C. Lange, and C. Rowat, editors, *AISB 2013*, pages 1–4, 2013. URL `http://www.cs.bham.ac.uk/research/projects/formare/events/aisb2013/proceedings.php`.

D. Kerbl. An Automated Induction Prover for Finite Sets Implemented in the Theorema System. Master's thesis, RISC, Johannes Kepler University Linz, October 2010.

P. Kügler and W. Windsteiger. *Algorithmische Methoden – Zahlen, Vektoren, Polynome*. Reihe: Mathematik kompakt. Birkhäuser Basel Boston Berlin, 1st edition, December 2008. ISBN 978-3-7643-8434-0. URL http://www.risc.uni-linz.ac.at/publications/books/AlgorithmischeMethoden/.

P. Kügler and W. Windsteiger. *Algorithmische Methoden – Funktionen, Matrizen, Multivariate Polynome*. Reihe: Mathematik kompakt. Birkhäuser Basel Boston Berlin, 1st edition, May 2012. ISBN 978-3-7643-8515-6. URL http://www.risc.jku.at/publications/books/AlgorithmischeMethoden/.

C. Lange, M. B. Caminati, M. Kerber, T. Mossakowski, C. Rowat, M. Wenzel, and W. Windsteiger. A Qualitative Comparison of the Suitability of Four Theorem Provers for Basic Auction Theory. In J. Carette, editor, *Conference on Intelligent Computer Mathematics (CICM 2013)*, volume 7961 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 200–215. Springer, 2013. ISBN 978-3-642-39319-8.

G. Mayrhofer, S. Saminger, and W. Windsteiger. CreaComp: Computer-Supported Experiments and Automated Proving in Learning and Teaching Mathematics. In E. Milkova, editor, *Proceedings of ICTMT8*, 2007a. ISBN 978-80-7041-285-5. 5 pages, distributed on CD.

G. Mayrhofer, S. Saminger, and W. Windsteiger. CreaComp: Experimental Formal Mathematics for the Classroom. In S. Li, D. Wang, and J.-Z. Zhang, editors, *Symbolic Computation and Education*, pages 94–114, Singapore, New Jersey, 2007b. World Scientific Publishing Co. ISBN 978-981-277-599-3. URL http://www.worldscibooks.com/socialsci/6642.html.

M. Sacha. Structuring and Reusing Knowledge in the Theorema System. Master's thesis, Internationaler Universitätslehrgang Informatics: Engineering and Management (ISI), Johannes Kepler University Linz, 2011.

W. Windsteiger. An Implementation of Rational Functions in PCL. Technical Report RISC-Linz Series 90-56, Univ. Linz, RISC, Linz, Austria, 1990a.

W. Windsteiger. An Approach to Object-Oriented Programming in C. Technical Report RISC-Linz Series 90-57, Univ. Linz, RISC, Linz, Austria, 1990b.

W. Windsteiger. Gröbner Bases: A Characterization by Syzygy Completeness and an Implementation. Master's thesis, RISC-Linz, University of Linz, Austria, 1992.

W. Windsteiger. GRÖBNER-IO: An Input/Output Libray for GRÖBNER. Technical Report in preparation, RISC-Linz, University of Linz, 1993a.

W. Windsteiger. Using GRÖBNER as a "Black Box". Technical Report 71, RISC-Linz, University of Linz, 1993b.

W. Windsteiger. GRÖBNER: A Library for Computing Gröbner Bases based on SACLIB. Talk given at the conference "Gröbner and Related Topics" in Dagstuhl, Germany, January 10-14 1994. URL http://www.risc.uni-linz.ac.at/people/wwindste/publications.html.

W. Windsteiger. Eine Einführung zur Methode der Gröbner Basen, September 26 1995a. Contributed talk at Treffen der ÖMG, Leoben, Austria.

W. Windsteiger. Mathematisches Problemlösen im Netz, July 5 1995b. Contributed talk at Telemedia'95, Hagenberg, Austria.

W. Windsteiger. Building Up Hierarchical Mathematical Domains Using Functors in THEOREMA. In A. Armando and T. Jebelean, editors, *Electronic Notes in Theoretical Computer Science*, volume 23-3, pages 83–102. Elsevier, 1999a. Calculemus 99 Workshop, Trento, Italy.

W. Windsteiger. THEOREMA: Overview on Using the System and Details on Composing Hierarchical Knowledge Bases. *School on Logic and Computation*, 1999b.

W. Windsteiger. Theorema: Ein Rahmen fuer Mathematik, Algorithmik und Didaktik. RISC Report Series 01-22, Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Schloss Hagenberg, 4232 Hagenberg, Austria, November 2001a. Course for mathematics teachers given in the frame of the "Tag der Mathematik 2001" at the University of Linz, November 23, 2001. In German.

W. Windsteiger. On a Solution of the Mutilated Checkerboard Problem using the Theorema Set Theory Prover. In S. Linton and R. Sebastiani, editors, *Proceedings of the Calculemus 2001 Symposium*, pages 28–47, 2001b.

W. Windsteiger. On a Solution of the Mutilated Checkerboard Problem using the Theorema Set Theory Prover, June 21 2001c. Contributed talk at Calculemus'2001, Siena, Italy.

W. Windsteiger. *A Set Theory Prover in Theorema: Implementation and Practical Applications*. PhD thesis, RISC Institute, May 2001d.

W. Windsteiger. A Set Theory Prover in *Theorema*. Technical Report 7, RISC, February 2001e. Talk given at the CAL'01 workshop, Las Palmas, Gran Canaria, February 20, 2001. Also available as SFB report 01-23.

W. Windsteiger. A Set Theory Prover in *Theorema*. In R. Moreno-Diaz, B. Buchberger, and J. Freire, editors, *Computer Aided Systems Theory*, number 2178 in LNCS, pages 525–539. Springer, 2001f. Proceedings of EUROCAST 2001 (8th International Conference on Computer Aided Systems Theory – Formal Methods and Tools for Computer Science), ISBN 3-540-42959.

W. Windsteiger. An Automated Prover for Set Theory in Theorema, June 5 2002a. Contributed talk at Calculemus'2002, Marseille, France.

W. Windsteiger. An Automated Prover for Zermelo-Fraenkel Set Theory in Theorema, October 20 2002b. Contributed talk at LMCS'02, Hagenberg, Austria.

W. Windsteiger. The Theorema System, September 27 2002c. Contributed talk at Calculemus Autumn School, Pisa, Italy.

W. Windsteiger. An Automated Prover for Set Theory in Theorema. In O. Caprotti and V. Sorge, editors, *Calculemus 2002, 10th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning: Work in Progress Papers*, pages 56–67, Marseille, France, June 2002d. ISBN 1427-4447 (ISSN). Seki-Report Series Nr. SR-02-04, Universität des Saarlandes.

W. Windsteiger. An Automated Prover for Zermelo-Fraenkel Set Theory in Theorema. In K. Nakagawa, editor, *Logic, Mathematics and Computer Science: Interactions (LMCS 2002)*, pages 266–280, RISC, Schloss Hagenberg, Austria, October 2002e. ISBN 3-902276-03-7. Symposium in Honor of Bruno Buchberger's 60th Birthday, RISC-report Series Nr. 02-60.

W. Windsteiger. Formalizing Mathematics / Computer-supported Mathematics, April 25 2003a. Contributed talk at SFB Statusseminar, Strobl, Austria.

W. Windsteiger. Exploring an Algorithm for Polynomial Interpolation in the Theorema System, September 12 2003b. Contributed talk at Calculemus'2003, Rome, Italy.

W. Windsteiger. An Automated Prover for Set Theory in Theorema, May 26 2003c. Contributed talk at Omega-Theorema Workshop, Hagenberg, Austria.

W. Windsteiger. Mathematica. In J. Grabmeier, E. Kaltofen, and V. Weispfenning, editors, *Computer Algebra Handbook: Foundations, Applications, Systems*, pages 314–320. Springer, 2003d. ISBN 3-540-65466-6.

W. Windsteiger. Exploring an Algorithm for Polynomial Interpolation in the Theorema System. In T. Hardin and R. Rioboo, editors, *Proceedings of the Calculemus 2003 Symposium*, pages 130–136, Rome Italy, September 2003e. Aracne Editrice S.R.L. ISBN 88-7999-545-6.

W. Windsteiger. Symbolic Solution Techniques for the Elastoplasticity Problem, March 31 2005a. Contributed talk at SFB Statusseminar 2005.

W. Windsteiger. CreaComp: Neue Möglichkeiten im e-learning für Mathematik, 22. April 2005b. Invited colloquium talk at Research Net Upper Austria: Brennpunkt Forschung.

W. Windsteiger. Theorema: A System for Mathematical Theory Exploration, April 26 2005c. Invited colloquium talk at Institute for Algebra, Charles University Prague.

W. Windsteiger. An Automated Theorem Prover for Set Theory within the Theorema System, April 25 2005d. Invited colloquium talk at Institute for Algebra, Charles University Prague.

W. Windsteiger. The CreaComp Project: Theorema for Computer-supported Teaching and Learning of Mathematics, November 14 2005e. URL `http://www.ags.uni-sb.de/~omega/workshops/TheoremaOmega05/`. Contributed talk at Theorema-Ultra-Omega'05 Workshop.

W. Windsteiger. Wie erfinde ich mathematische Algorithmen? Wie beweise ich mathematische Algorithmen? RISC Report Series 05-18, Research Institute for Symbolic Computation (RISC), Johannes Kepler University of Linz,

Schloss Hagenberg, 4232 Hagenberg, Austria, December 2005f. Presentation slides for a presentation given at Schwerpunktfach Mathematik, Europagymnasium Auhof, December 15, 2005.

W. Windsteiger. An Automated Prover for Zermelo-Fraenkel Set Theory in Theorema. *JSC*, 41(3-4):435–470, 2006a. URL `http://authors.elsevier.com/sd/article/S0747717105001495`.

W. Windsteiger. The Theorema System, February 20 2006b. Invited colloquium talk at Carnegie Mellon University, Computer Science seminar.

W. Windsteiger. Computer-supported Proving in ZF Set Theory with the Theorema System, March 2 2006c. Invited colloquium talk at Carnegie Mellon University, Math Logic seminar.

W. Windsteiger. Introduction to Theorema: An Example of a Formal Math System, March 6 2006d. Contributed talk at Special Semester on Gröbner Bases: Workshop on Formal Gröbner Bases Theory. RICAM, Linz.

W. Windsteiger. Introduction to the Gröbner Bases Method, April 28 2006e. Talk given in the frame of the seminar "Fast SAT Solvers and Practical Decision Procedures". Invited colloquium talk at Carnegie Mellon University, Computer Science Deptartment.

W. Windsteiger. Analytica V: Towards the Mordell-Weil Theorem, July 9 2006f. Contributed talk at Calculemus'06.

W. Windsteiger. Towards Computer-Supported Proving in Maths Education, June 21 2007a. URL `http://matserv.pmmf.hu/cadgme/`. Contributed talk at First Central- and Eastern European Conference on Computer Algebra- and Dynamic Geometry Systems in Mathematics Education (CADGME'07).

W. Windsteiger. CreaComp: Computer-Supported Experiments and Automated Proving in Learning and Teaching Mathematics, July 3 2007b. URL `http://www.ictmt8.org/ictmt8/`. Contributed talk at 8th International Conference on Technology in Mathematics Teaching (ICTMT8).

W. Windsteiger. Stimulating Students' Creativity Through Computer-Supported Experiments and Automated Theorem Proving. In E. Velikova and A. Andzans, editors, *Promoting Creativity for all Students in Mathematics Education*, pages 351–357, 2008a. ISBN 978-954-712-420-2. Proceedings of Discussion Group 9, the 11th International Congress on Mathematical Education (ICME 11), Monterrey, Mexico, July 7–13.

W. Windsteiger. Theorema: Automated Theorem Proving Meets Teaching of Mathematics, July 9 2008b. Contributed talk at International Congress on Mathematical Education, ICME 11, JEM Workshop.

W. Windsteiger. Theorema: A System for Computer-Supported Theorem Proving and Theory Development based on Mathematica, November 14 2009. URL `http://www.wolfram.com/services/seminars/mideastconf2009/`. Invited talk at Middle East Mathematica Conference 2009.

W. Windsteiger. Theorema 2: Some Design Considerations for the Re-Implementation of the Theorema System, August 5 2010. URL http://www.risc.jku.at/about/conferences/ciao2010/. Contributed talk at CIAO 2010 Workshop.

W. Windsteiger. Using Theorema in the Formalization of Theoretical Economics, April 5 2011. Contributed talk at CIAO 2011.

W. Windsteiger. Theorema 2.0: Current Status of the Implementation, April 18 2012a. URL http://www.chalmers.se/cse/EN/organization/divisions/software-technology/ciao-workshop. Contributed talk at CIAO 2012.

W. Windsteiger. Theorema 2.0: A Graphical User Interface for a Mathematical Assistant System. In C. Kaliszyk and C. Lüth, editors, *Proceedings 10th International Workshop On User Interfaces for Theorem Provers, Bremen, Germany, July 11th 2012*, volume 118 of *Electronic Proceedings in Theoretical Computer Science*, pages 72–82. Open Publishing Association, 2012b. ISBN 2075-2180 (ISSN). doi: 10.4204/EPTCS.118.5. URL http://arxiv.org/abs/1307.1945v1.

W. Windsteiger. Theorema 2.0: A Graphical User Interface for a Mathematical Assistant System. In J. Davenport, J. Jeuring, C. Lange, and P. Libbrecht, editors, *24th OpenMath Workshop, 7th Workshop on Mathematical User Interfaces (MathUI), and Intelligent Computer Mathematics Work in Progress*, number 921 in CEUR Workshop Proceedings, pages 73–81, Aachen, 2012c. ISBN 1613-0073 (ISSN). URL http://ceur-ws.org/Vol-921/.

W. Windsteiger. Theorema 2.0: A Graphical User Interface for a Mathematical Assistant System, July 11 2012d. URL http://www.informatik.uni-bremen.de/uitp12/. Contributed talk at UITP 2012.

W. Windsteiger. Theorema 2.0: Automated and Interactive Theorem Proving in Math Education, June 10 2013a. Contributed talk at CSASC'2013, Koper, Slovenia.

W. Windsteiger. Theorema 2.0: An Open-Source Mathematical Assistant System for Automated and Interactive Reasoning, October 24 2013b. URL http://pas2013.cc4cm.org/. Invited talk at PAS'2013: Second International Seminar on Program Verification, Automated Debugging and Symbolic Computation.

W. Windsteiger and B. Buchberger. GRÖBNER: A Library for Computing Gröbner Bases based on SACLIB. Technical Report 72, RISC-Linz, University of Linz, 1993.

W. Windsteiger, B. Buchberger, and M. Rosenkranz. Theorema. In F. Wiedijk, editor, *The Seventeen Provers of the World*, volume 3600 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 96–107. Springer Berlin Heidelberg New York, 2006. ISBN 3-540-30704-4. URL http://link.springer.com/book/10.1007/11542384.