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*

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.

Key words: Mathematical assistant, automated reasoning, theory exploration, “Lazy Thinking”, Theorema

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,

* Corresponding author.
Email address: `Temur.Kutsia@risc.uni-linz.ac.at` (Temur Kutsia).

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 [20,19].

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 [20,19]. 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.

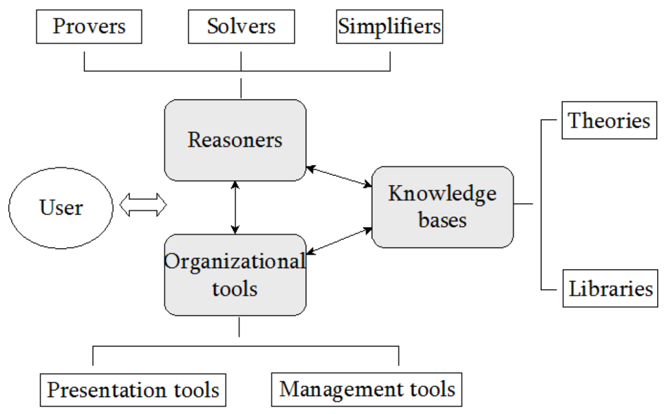


Fig. 1. *Theorema* components for theory exploration.

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 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

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

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 [20,19], include: the prover for classical predicate logic that implements a sequent calculus with meta-variables; 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 $\underset{x}{\forall}\, Q[x, A[x]]$, 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

Problem[“Gröbner Bases”,

$$\underset{F}{\forall} \bigwedge \begin{cases} \text{is-finite}[\mathrm{GB}[F]] \\ \text{is-Gröbner-basis}[\mathrm{GB}[F]] \\ \text{ideal}[F] = \text{ideal}[\mathrm{GB}[F]] \end{cases}]$$

(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

$$\textbf{Theory}[\text{"Gröbner Bases prerequisites"},$$

$$\underset{G}{\forall}\ \text{is-Gröbner-basis}[G] \Leftrightarrow \text{is-confluent}[\rightarrow_G]$$

$$\underset{G,h_1,h_2}{\forall}\ h_1 \rightarrow_G h_2 \Leftrightarrow \underset{g\in G}{\exists} \bigwedge \begin{cases} \mathrm{lp}[g] \mid \mathrm{lp}[h_1] \\ h_2 = h_1 - (\mathrm{lm}[h_1]/\mathrm{lm}[g])g \end{cases}]$$

$$\ldots$$

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*:

$$\mathrm{GB}[F] = \mathrm{GB}[F, \mathrm{pairs}[F]]$$

$$\mathrm{GB}[F, \langle\rangle] = F$$

$$\mathrm{GB}[F, \langle\langle g_1, g_2\rangle, \overline{p}\rangle] =$$

$$\text{where}[f = lc[g_1, g_2], h_1 = \mathrm{trd}[\mathrm{rd}[f, g_1], F], h_2 = \mathrm{trd}[\mathrm{rd}[f, g_2], F],$$

$$\begin{cases} \mathrm{GB}[F, \langle\overline{p}\rangle] & \Leftarrow h_1 = h_2 \\ \mathrm{GB}[F \frown df[h_1, h_2],\ \langle\overline{p}\rangle \asymp \Big\langle \langle F_k, df[h_1,h_2]\rangle_{\underset{k=1,\ldots,|F|}{|}} \Big\rangle] & \Leftarrow \text{otherwise} \end{cases}]$$

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 $\overline{p}$, 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

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

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", $\neg\,\mathrm{rat}[\sqrt{2}]$]

using the knowledge about positive real numbers

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

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

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

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

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

$$(6)\quad \mathrm{coprime}[\mathrm{a}_0, \mathrm{b}_0] \wedge \mathrm{nat}[\mathrm{a}_0] \wedge \mathrm{nat}[\mathrm{b}_0] \wedge \sqrt{2} = \frac{\mathrm{a}_0}{\mathrm{b}_0}$$

for arbitrary but fixed a_0 and b_0. Now, nat$[\mathrm{a}_0]$ and nat$[\mathrm{b}_0]$ are used to instantiate Lemma "coprime", and $\sqrt{2} = \frac{\mathrm{a}_0}{\mathrm{b}_0}$ 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 $\sqrt{2} = \frac{\mathrm{a}_0}{\mathrm{b}_0}$ results in

$$(9)\quad 2 * {\mathrm{b}_0}^2 = {\mathrm{a}_0}^2,$$

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

$$(10)\quad \neg\, \mathrm{coprime}[\mathrm{a}_0, \mathrm{b}_0],$$

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.

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: $\underset{\overline{x},\overline{y},\overline{z},u}{\forall} f[\overline{x}, u, \overline{y}, u, \overline{z}] = f[\overline{x}, u, \overline{y}, \overline{z}]$ for idempotence and $\underset{\overline{x},\overline{y},\overline{z}}{\forall} f[\overline{x}, f[\overline{y}], \overline{z}] = f[\overline{x}, \overline{y}, \overline{z}]$ 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.

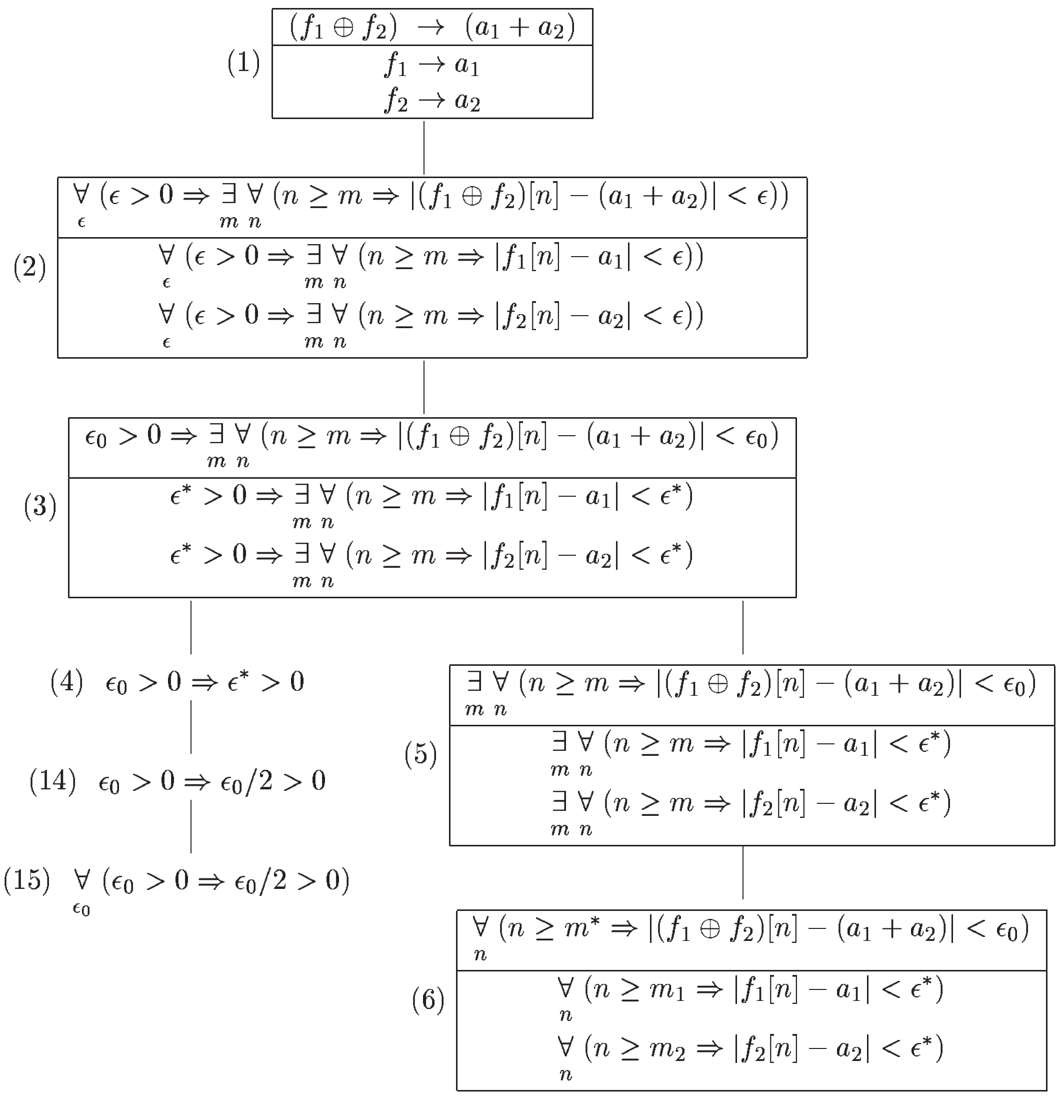
Proofs are described by the Proof Communication Language PCL [31]. They are structured into lemmata/propositions. Proofs of universally closed theorems are displayed as equational chains, while those of existential theorems represent sequences of equations. In failing proofs, on the one hand, the theorems 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.



(1) $(f_1 \oplus f_2) \to (a_1 + a_2)$
$f_1 \to a_1$
$f_2 \to a_2$
(2) $\underset{\epsilon}{\forall} (\epsilon > 0 \Rightarrow \underset{m}{\exists} \underset{n}{\forall} (n \geq m \Rightarrow |(f_1 \oplus f_2)[n] - (a_1 + a_2)| < \epsilon))$
$\underset{\epsilon}{\forall} (\epsilon > 0 \Rightarrow \underset{m}{\exists} \underset{n}{\forall} (n \geq m \Rightarrow |f_1[n] - a_1| < \epsilon))$
$\underset{\epsilon}{\forall} (\epsilon > 0 \Rightarrow \underset{m}{\exists} \underset{n}{\forall} (n \geq m \Rightarrow |f_2[n] - a_2| < \epsilon))$
(3) $\epsilon_0 > 0 \Rightarrow \underset{m}{\exists} \underset{n}{\forall} (n \geq m \Rightarrow |(f_1 \oplus f_2)[n] - (a_1 + a_2)| < \epsilon_0)$
$\epsilon^* > 0 \Rightarrow \underset{m}{\exists} \underset{n}{\forall} (n \geq m \Rightarrow |f_1[n] - a_1| < \epsilon^*)$
$\epsilon^* > 0 \Rightarrow \underset{m}{\exists} \underset{n}{\forall} (n \geq m \Rightarrow |f_2[n] - a_2| < \epsilon^*)$
(4) $\epsilon_0 > 0 \Rightarrow \epsilon^* > 0$
(14) $\epsilon_0 > 0 \Rightarrow \epsilon_0/2 > 0$
(15) $\underset{\epsilon_0}{\forall} (\epsilon_0 > 0 \Rightarrow \epsilon_0/2 > 0)$
(5) $\underset{m}{\exists} \underset{n}{\forall} (n \geq m \Rightarrow |(f_1 \oplus f_2)[n] - (a_1 + a_2)| < \epsilon_0)$
$\underset{m}{\exists} \underset{n}{\forall} (n \geq m \Rightarrow |f_1[n] - a_1| < \epsilon^*)$
$\underset{m}{\exists} \underset{n}{\forall} (n \geq m \Rightarrow |f_2[n] - a_2| < \epsilon^*)$
(6) $\underset{n}{\forall} (n \geq m^* \Rightarrow |(f_1 \oplus f_2)[n] - (a_1 + a_2)| < \epsilon_0)$
$\underset{n}{\forall} (n \geq m_1 \Rightarrow |f_1[n] - a_1| < \epsilon^*)$
$\underset{n}{\forall} (n \geq m_2 \Rightarrow |f_2[n] - a_2| < \epsilon^*)$

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

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:

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

For brevity, the type information is not included.

The proof tree is presented in Fig. 2 and Fig. 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 sit-

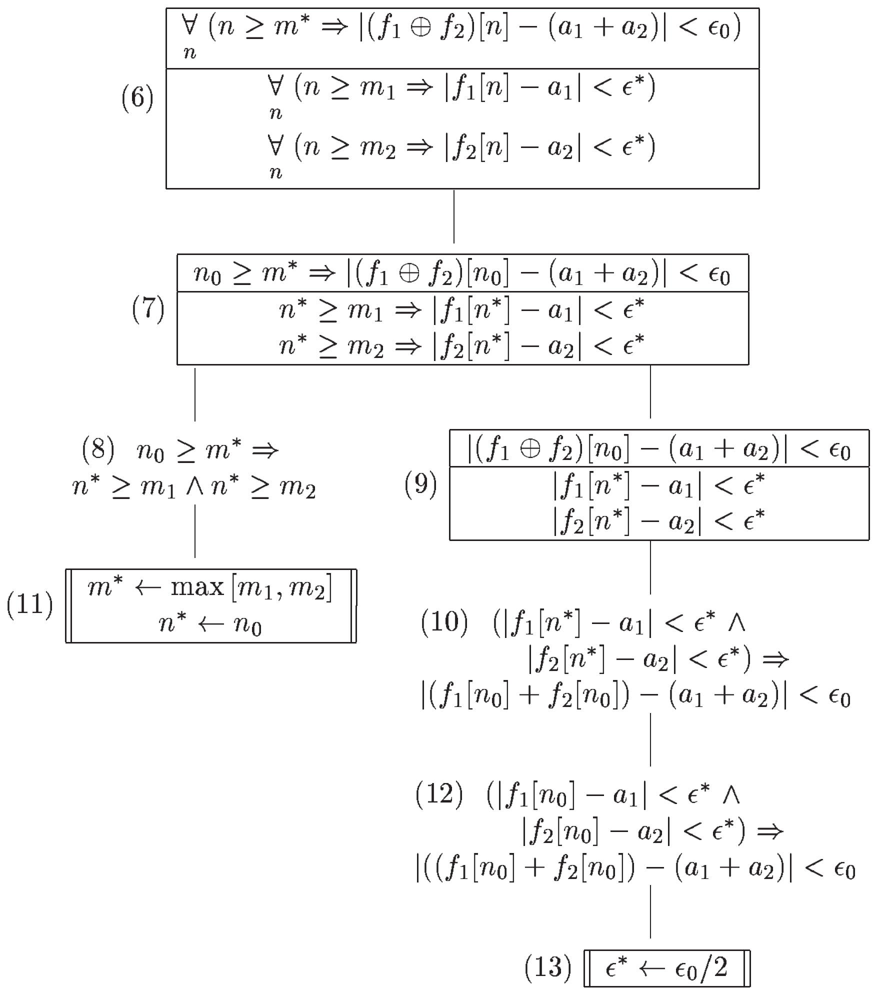

(6)
$\forall_n (n \geq m^* \Rightarrow |(f_1 \oplus f_2)[n] - (a_1 + a_2)| < \epsilon_0)$
$\forall_n (n \geq m_1 \Rightarrow |f_1[n] - a_1| < \epsilon^*)$
$\forall_n (n \geq m_2 \Rightarrow |f_2[n] - a_2| < \epsilon^*)$
(7)
$n_0 \geq m^* \Rightarrow |(f_1 \oplus f_2)[n_0] - (a_1 + a_2)| < \epsilon_0$
$n^* \geq m_1 \Rightarrow |f_1[n^*] - a_1| < \epsilon^*$
$n^* \geq m_2 \Rightarrow |f_2[n^*] - a_2| < \epsilon^*$
(8) $n_0 \geq m^* \Rightarrow n^* \geq m_1 \wedge n^* \geq m_2$
(9)
$|(f_1 \oplus f_2)[n_0] - (a_1 + a_2)| < \epsilon_0$
$|f_1[n^*] - a_1| < \epsilon^*$
$|f_2[n^*] - a_2| < \epsilon^*$
(11)
$m^* \leftarrow \max[m_1, m_2]$
$n^* \leftarrow n_0$
(10) $(|f_1[n^*] - a_1| < \epsilon^* \wedge |f_2[n^*] - a_2| < \epsilon^*) \Rightarrow |(f_1[n_0] + f_2[n_0]) - (a_1 + a_2)| < \epsilon_0$
(12) $(|f_1[n_0] - a_1| < \epsilon^* \wedge |f_2[n_0] - a_2| < \epsilon^*) \Rightarrow |((f_1[n_0] + f_2[n_0]) - (a_1 + a_2)| < \epsilon_0$
(13) $\epsilon^* \leftarrow \epsilon_0/2$

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

uation (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.)

$$\underset{x}{\forall} P_1[x], \underset{x}{\forall} P_2[x] \ \vdash \ \underset{x}{\forall} P_0[x] \quad \longmapsto \quad P_1[x_1^*], P_2[x_2^*] \ \vdash \ P_0[x_0] \qquad (\forall)$$

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.

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

$$Q_1 \Rightarrow P_1,\ Q_2 \Rightarrow P_2 \ \vdash \ Q_0 \Rightarrow P_0 \quad \longmapsto \quad \begin{cases} Q_0 \Rightarrow Q_1 \wedge Q_2 \\ P_1, P_2 \ \vdash \ P_0 \end{cases} \tag{$\Rightarrow$}$$

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:

$$\underset{x}{\exists} P_1[x], \underset{x}{\exists} P_2[x] \ \vdash \ \underset{x}{\exists} P_0[x] \quad \longmapsto \quad P_1[x_1], P_2[x_2] \ \vdash \ P_0[x^*] \tag{$\exists$}$$

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:

$$Q_1 \wedge P_1,\ Q_2 \wedge P_2 \ \vdash \ Q_0 \wedge P_0 \quad \longmapsto \quad \begin{cases} Q_1 \wedge Q_2 \Rightarrow Q_0 \\ P_1, P_2 \ \vdash \ P_0 \end{cases} \tag{$\wedge$}$$

Similarly to the rule ($\Rightarrow$), 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 ($\exists$), ($\wedge$) are applied in one step, as are also the rules ($\forall$), ($\Rightarrow$).

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 ($\forall$) to (6), and then the rule ($\Rightarrow$) 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) $\underset{k,i,j}{\forall} (k \geq \max[i,j] \Rightarrow k \geq i \wedge k \geq j)$,

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

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.

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, page 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 $u = \int_0 1\, g(x, \xi)\, f(\xi) d\xi$. In the above example, one has

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

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 $T\, u = c_0\, u^{(n)} + \cdots + c_{n-1}\, u' + c_n\, u$, 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

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

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

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

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 $u \mapsto (x \mapsto \int_a^x u(\xi)\,d\xi)$ by A, its dual $u \mapsto (x \mapsto \int_x^b u(\xi)\,d\xi)$ 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 $G\,T = 1 - P$, 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 $L\,u = 0$ and $R\,u = 0$ on the interval $[0, \pi]$. Using the *Theorema* command

Compute[Green[$D2 + 2D + 1, \langle L, R\rangle$, by → GreenEvaluator]

we get the output

$$(1 - \pi^{-1})\lceil e^{-x}x\rceil A\lceil e^{x}\rceil - \lceil e^{-x}\rceil A\lceil e^{x}x\rceil + \pi^{-1}\lceil e^{-x}x\rceil A\lceil e^{x}x\rceil \\ - \pi^{-1}\lceil e^{-x}x\rceil B\lceil e^{x}\rceil + \pi^{-1}\lceil e^{-x}x\rceil B\lceil e^{x}x\rceil.$$

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 [94,23], 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 (AreaCad 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 $r \leq R/2$."*

We prepare the following input to Theorema:

$$
\begin{aligned}
&\textbf{Proposition}[\text{“Tri”}, \text{any}[O, A, B, C, P, X, Y, O_1],\\
&\quad \text{incircle}[O, A, B, C, P] \wedge \text{midpoint}[X, A, B] \wedge \text{midpoint}[Y, A, C] \wedge\\
&\quad \text{inter}[O_1, \text{tline}[X, A, B], \text{tline}[Y, A, C]] \wedge \text{circle}[O_1, A] \qquad\qquad].\\
&\quad \Rightarrow 4 \cdot \text{seglength}[O, P]^2 \leq \text{seglength}[O_1, A]^2]
\end{aligned}
$$

By the command

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

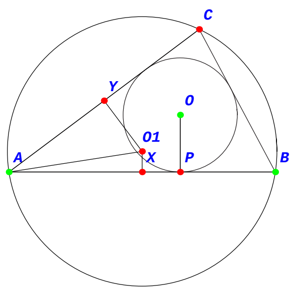


Fig. 4. Example problem for geometry prover.

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

Prove[Proposition["Tri"], by → GeometryProver,
    ProverOptions → {Method → 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:*

$$\begin{array}{ll}
\{A, B, O\} \text{ free points} & \\
\alpha_A B \perp AO,\ \alpha_A \in AO & (\text{nondegenerate condition } A \neq O) \\
\alpha 1_A B \parallel B\alpha_A,\ \frac{\overline{B\alpha 1_A}}{\overline{B\alpha_A}} = 2 & (\text{ndg. cond. } B \neq \alpha_A) \\
\alpha_B A \perp BO,\ \alpha_B \in BO & (\text{ndg. cond. } B \neq O) \\
\alpha 1_B A \parallel A\alpha_B,\ \frac{\overline{A\alpha 1_B}}{\overline{A\alpha_B}} = 2 & (\text{ndg. cond. } A \neq \alpha_B) \\
C = A\alpha 1_A \cap B\alpha 1_B, & (\text{ndg. cond. } A \neq \alpha 1_A, B \neq \alpha 1_B, A\alpha 1_A \nparallel B\alpha 1_B) \\
PO \perp AB,\ P \in AB & (\text{ndg. cond. } A \neq B) \\
\alpha_C O \perp AC,\ \alpha_C \in AC & (\text{ndg. cond. } A \neq C) \\
XA \parallel AB,\ \frac{\overline{AX}}{\overline{AB}} = \frac{1}{2} & (\text{ndg. cond. } A \neq B) \\
YA \parallel AC,\ \frac{\overline{AY}}{\overline{AC}} = \frac{1}{2} & (\text{ndg. cond. } A \neq C) \\
Y\gamma_{O_1} \perp YA,\ \frac{\overline{Y\gamma_{O_1}}}{\overline{YA}} = r_{23} & (\text{ndg. cond. } Y \neq A) \\
X\beta_{O_1} \perp XA,\ \frac{\overline{X\beta_{O_1}}}{\overline{XA}} = r_{22} & (\text{ndg. cond. } X \neq A) \\
O_1 = X\beta_{O_1} \cap Y\gamma_{O_1} & (\text{ndg. cond. } X \neq \alpha 1_A\beta_{O_1}, Y \neq \gamma_{O_1}, \beta_{O_1}X \nparallel \gamma_{O_1}Y)
\end{array}$$

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 $-\overline{AO_1}^2 + 4\overline{PO}^2 \leq 0$.

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 Subsection 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[$\downarrow x, \downarrow y, \uparrow rem, \uparrow quo$],

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

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[$\downarrow x, \downarrow y, \uparrow rem, \uparrow quo$],

$$\begin{array}{l} \text{Pre} \rightarrow ((x \geq 0) \wedge (y > 0)), \\ \text{Post} \rightarrow ((quo * y + rem = x) \wedge (0 \leq rem < y)) \end{array}\ ].$$

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 Subsection 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 *generat-*

ing 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:

$$\underset{x:I_F[x]}{\forall} (Q[x] \Rightarrow I_S[x]) \qquad \underset{x:I_F[x]}{\forall} (\neg Q[x] \Rightarrow I_F[R[x]])$$

$$\underset{x:I_F[x]}{\forall} (\neg Q[x] \Rightarrow I_R[x]) \qquad \underset{x:I_F[x]}{\forall} (\neg Q[x] \Rightarrow \underset{y}{\forall} (O_F[R[x], y] \Rightarrow I_C[x, y]))$$

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

$$\underset{x:I_F[x]}{\forall} (Q[x] \Rightarrow O_F[x, S[x]])$$

$$\underset{x:I_F[x]}{\forall} (\neg Q[x] \Rightarrow \underset{y}{\forall} (O_F[R[x], y] \ \Rightarrow O_F[x, C[x, y]]))$$

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.

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

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].

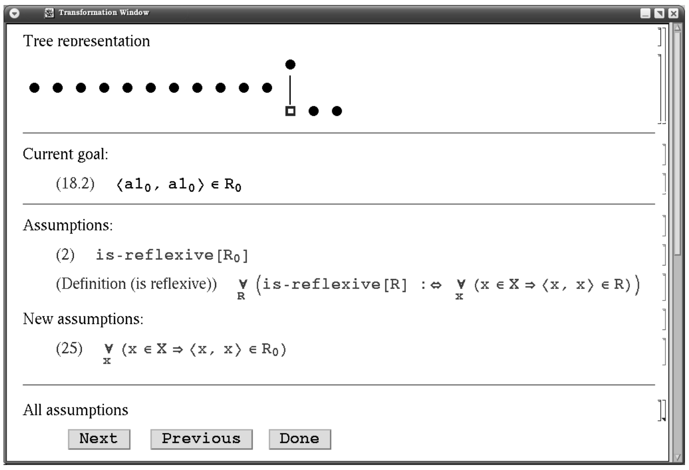
4 Organizational Tools

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



Transformation Window

Tree representation

Current goal:

(18.2) $\langle al_0, al_0 \rangle \in R_0$

Assumptions:

(2) is-reflexive$[R_0]$

(Definition (is reflexive)) $\underset{R}{\forall}\left(\text{is-reflexive}[R] :\Leftrightarrow \underset{x}{\forall}(x \in X \Rightarrow \langle x, x \rangle \in R)\right)$

New assumptions:

(25) $\underset{x}{\forall}(x \in X \Rightarrow \langle x, x \rangle \in R_0)$

All assumptions

Fig. 5. A Focus Window

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 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.

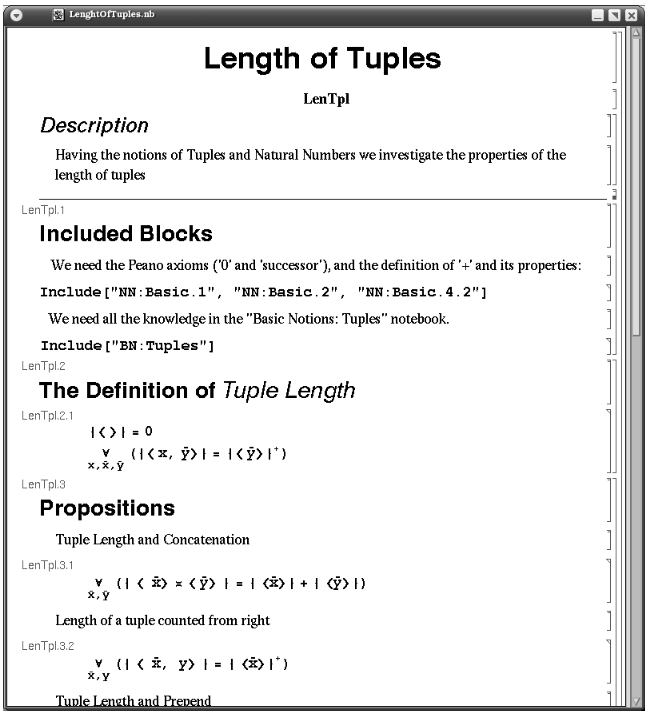
The focus windows presentation technique can be used also for incomplete or incorrect proofs, making it also a useful tool for prover debugging. Details of the technique can be found in [67].

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


LenghtOfTuples.nb

Length of Tuples

LenTpl

Description

Having the notions of Tuples and Natural Numbers we investigate the properties of the length of tuples

LenTpl.1

Included Blocks

We need the Peano axioms ('0' and 'successor'), and the definition of '+' and its properties:

```
Include["NN:Basic.1", "NN:Basic.2", "NN:Basic.4.2"]
```

We need all the knowledge in the "Basic Notions: Tuples" notebook.

```
Include["BN:Tuples"]
```

LenTpl.2

The Definition of *Tuple Length*

LenTpl.2.1

$|\langle\rangle| = 0$

$\underset{x,\bar{x},\bar{y}}{\forall} (|\langle x, \bar{y}\rangle| = |\langle\bar{y}\rangle|^{+})$

LenTpl.3

Propositions

Tuple Length and Concatenation

LenTpl.3.1

$\underset{\bar{x},\bar{y}}{\forall} (|\langle \bar{x}\rangle \asymp \langle\bar{y}\rangle| = |\langle\bar{x}\rangle| + |\langle\bar{y}\rangle|)$

Length of a tuple counted from right

LenTpl.3.2

$\underset{\bar{x},y}{\forall} (|\langle \bar{x}, y\rangle| = |\langle\bar{x}\rangle|^{+})$

Tuple Length and Prepend

Fig. 6. A *Theorema* notebook.

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 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:

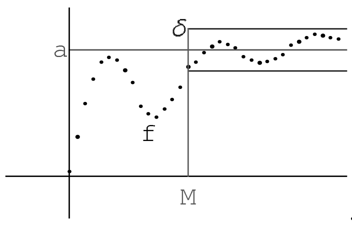
Prove[“LenTpl.3.1”, using $\rightarrow$ ⟨“BN:Tuples”, “NN:Basic”⟩, by $\rightarrow$ ...].

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

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

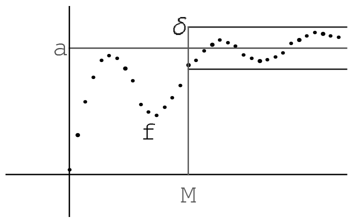
In *Theorema,* we may now design a new graphical symbol for this notion by using the graphics tools of Mathematica (or just by taking a hand-drawing and making it a Mathematica object) and equip it with slots (boxes) for the arguments, like, for example, the new symbol



.

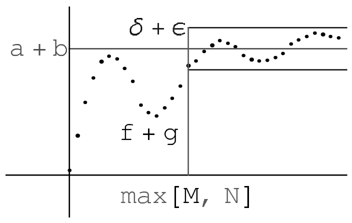
Now, by the following declaration

Logicographic-Declaration[“limit symbol”, any$[f, a, \delta, M]$,



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

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 Subsection 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, and I. Schena. Mathematical knowledge management in HELM. *Annals of Mathematics and Artificial Intelligence*, 38(1):27–46, 2003.

[2] L. Bachmair, N. Dershowitz, and D. Plaisted. Completion without failure. In H. Aït-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 2, pages 1–30. Elsevier Science, 1989.

[3] D. Basin, Y. Deville, P. Flener, A. Hamfelt, and J. F. Nilsson. Synthesis of programs in computational logic. In M. Bruynooghe and K.-K. Lau, editors, *Program Development in Computational Logic*, volume 3049 of *LNCS*, pages 30–65. Springer, 2004.

[4] Y. Bertot and P. Castèran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[5] W. Bledsoe. Challenge problems in elementary analysis. *J. Automated Reasoning*, 6:341–359, 1990.

[6] A. Buch and 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, editor, *Multidimensional Systems Theory*, pages 184–232. Reidel Publishing Company, Dodrecht, 1985.

[9] B. Buchberger. Symbolic computation: Computer algebra and logic. In F. Baader and K. U. Schulz, editors, *Frontiers of Combining Systems*, volume 3 of *Applied Logic Series*, pages 193–219. Kluwer Academic Publishers, 1996.

[10] B. Buchberger. Introduction to Gröbner Bases. In [21], pages 3–31, 1998.

[11] B. Buchberger. Theory exploration versus theorem proving. In A. Armando and T. Jebelean, editors, *Proc. of Calculemus'99*, volume 23 of *ENTCS*. Elsevier, 1999.

[12] B. Buchberger. Theory exploration with *Theorema*. *Analele Universităţii din Timişoara, Ser. Matematică-Informatică*, XXXVIII(2):9–32, 2000.

[13] B. Buchberger. Algorithm invention and verification by Lazy Thinking. In [65], pages 2–26, 2003.

[14] B. Buchberger. Algorithm-supported mathematical theory exploration: A personal view and strategy. In [16], pages 236–250, 2004.

[15] B. Buchberger. Towards the automated synthesis of a Gröbner bases algorithm. *RACSAM (Review of the Spanish Royal Academy of Sciences)*, 98(1):65–75, 2004.

[16] B. Buchberger and J. A. Campbell, editors. *Proc. of the 7th Int. Conf. on Artificial Intelligence and Symbolic Computation, AISC'04*, volume 3249 of *LNAI*, Hagenberg, Austria, 2004. Springer.

[17] B. Buchberger and A. Crăciun. Algorithm synthesis by Lazy Thinking: Examples and implementation in *Theorema*. In F. Kamareddine, editor, *Proc. of the Mathematical Knowledge Management Workshop, Edinburgh*, volume 93 of *ENTCS*, pages 24–59. Elsevier, 2003.

[18] B. Buchberger and A. Crăciun. Algorithm synthesis by Lazy Thinking: Using problem schemes. In [66], pages 90–106, 2004.

[19] B. Buchberger, C. Dupré, T. Jebelean, F. Kriftner, K. Nakagawa, D. Văsaru, and W. Windsteiger. The *Theorema* project: A progress report. In [44], pages 98–113, 2000.

[20] B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuţa, and D. Văsaru. A survey of the *Theorema* project. In W. Küchlin, editor, *Proc. of the Int. Symposium on Symbolic and Algebraic Computation ISSAC'97*, pages 384–391. ACM Press, 1997.

[21] B. Buchberger and F. Winkler. *Gröbner Bases and Applications*. Cambridge University Press, Cambridge, UK, 1998. Proc. of the Int. Conf. "33 Years of Gröbner Bases", 1998, RISC, Austria, London Mathematical Society Lecture Note Series, Vol. 251.

[22] O. Caprotti and D. Carlisle. OpenMath and MathML: Semantic mark up for mathematics. *ACM Crossroads, Special Issue on Markup Languages*, 6(2), 1999. ACM Press.

[23] S. C. Chou. *Mechanical Geometry Theorem Proving.* Reidel, Dordrecht Boston, 1975.

[24] S. C. Chou, X. S. Gao, and J. Z. Zhang. Automated production of traditional proofs in euclidian geometry. In *Proc. of 8th IEEE Symposium on Logic in Computer Science*, pages 48–56. IEEE Computer Society Press, 1993.

[25] E. A. Coddington and 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*, volume 33 of *LNCS*, pages 134–183. Springer, 1975.

[27] R. Constable. *Implementing Mathematics Using the Nuprl Proof Development System.* Prentice Hall, 1986.

[28] A. Crăciun and B. Buchberger. Preprocessed Lazy Thinking: Synthesis of sorting algorithms. Technical Report 04–17, RISC, Linz, Austria, 2004.

[29] J. W. de Bakker and 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 and 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, 1976.

[33] B. Elspas, M. W. Green, K. N. Lewitt, and R. J. Waldinger. Research in interactive program—proving techniques. Technical report, Stanford Research Institute, Menlo Park, California, USA, May 1972.

[34] P. Flener, K.-K. Lau, M. Ornaghi, and J. D. C. Richardson. An abstract formalization of correct schemas for program synthesis. *J. of Symbolic Computation*, 30(1):93–127, 2000.

[35] H. Ganzinger, editor. *Proc. of the 16th Int. Conf. on Automated Deduction, CADE'99*, volume 1632 of *LNAI*, Trento, Italy, 1999. Springer.

[36] R. W. Gosper. Decision procedures for indefinite hypergeometric summation. *Proc. of the National Academy of Science, USA*, 75(5–6):40–42, 1978.

[37] J. Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, UK, 1995.

[38] K. Hodgson and 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 and A. Bundy. Productive use of failure in inductive proof. *J. Automated Reasoning*, 16(1–2):79–111, 1996.

[40] T. Jebelean. Natural proofs in elementary analysis by S-Decomposition. Technical Report 01-33, RISC, Linz, Austria, 2001.

[41] T. Jebelean, L. Kovács, and 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*, volume 1. Teubner, Stuttgart, 10th edition, 1983.

[43] D. Kapur. Using Gröbner Bases to reason about geometry problems. *J. Symbolic Computation*, 2:399–408, 1986.

[44] M. Kerber and M. Kohlhase, editors. *Proc. of Calculemus'2000*, St. Andrews, UK, 2000.

[45] M. Kirchner. Program verification with the mathematical software system *Theorema.* Technical Report 99–16, RISC, Linz, Austria, 1999.

[46] M. Kohlhase. OMDoc: An infrastructure for OpenMath content dictionary information. *ACM SIGSAM Bulletin*, 34(2):43–48, 2000.

[47] B. Konev and T. Jebelean. Using meta-variables for natural deduction in *Theorema.* In [44], pages 160–175, 2000.

[48] L. Kovács and T. Jebelean. Automated generation of loop invarinats by recurrence solving in *Theorema.* In [66], 2004.

[49] I. Kraan, D. Basin, and A. Bundy. Middle-out reasoning for synthesis and induction. *J. Automated Reasoning*, 16(1–2):113–145, 1996.

[50] A. M. Krall. *Applied Analysis.* D. Reidel Publishing Company, Dordrecht, 1986.

[51] T. Kutsia. Equational prover of *Theorema.* In R. Nieuwenhuis, editor, *Proc. of the 14th Int. Conf. on Rewriting Techniques and Applications, RTA'03*, volume 2706 of *LNCS*, pages 367–379, Valencia, Spain, 2003. Springer.

[52] T. Kutsia. Solving equations involving sequence variables and sequence functions. In [16], pages 157–170, 2004.

[53] T. Kutsia and B. Buchberger. Predicate logic with sequence variables and sequence function symbols. In A. Asperti, G. Bancerek, and A. Trybulec, editors, *Proc. of the 3rd Int. Conf. on Mathematical Knowledge Management*, volume 3119 of *LNCS*, pages 205–219. Springer, 2004.

[54] T. Kutsia and K. Nakagawa. An interface between *Theorema* and external automated deduction systems. Technical Report 00–29, RISC, Linz, 2000.

[55] B. Kutzler and S. Stifter. On the application of Buchberger's Algorithm to automated geometry theorem proving. *J. Symbolic Computation*, 2:389–397, 1986.

[56] R. Letz, J. Schumann, S. Bayerl, and W. Bibel. Setheo: A high-performance theorem prover. *J. Automated Reasoning*, 8(2):183–212, 1992.

[57] J. Loeckx and K. Sieber. *The Foundations of Program Verification.* Teubner, second edition, 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 and J. Siekmann. Knowledge-based proof planning. *Artificial Intelligence*, 115(1):65–105, 1999.

[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, editor. *Generalized Inverses and Applications*, Proc. of an Advanced Seminar Sponsored by the Mathematics Research Center, New York, 1976. Academic Press.

[64] P. Paule and M. Schorn. A Mathematica version of Zeilberger's algorithm for proving binomial coefficient identities. *J. Symbolic Computation*, 20(5–6):673–698, 1995.

[65] D. Petcu, V. Negru, D. Zaharie, and T. Jebelean, editors. *Proc. of SYNASC'03, 5th Int. Workshop on Symbolic and Numeric Algorithms for Scientific Computing*, Timişoara, Romania, 2003. Mirton.

[66] D. Petcu, V. Negru, D. Zaharie, and T. Jebelean, editors. *Proc. of SYNASC'04, 6th Int. Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, Timişoara, Romania, 2004. Mirton.

[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 and B. Buchberger. An environment for building mathematical knowledge libraries. In C. Benzmüller and W. Windsteiger, editors, *Proc. of the first Workshop on Computer-Supported Mathematical Theory Development, IJCAR'04*, pages 19–29, Cork, Ireland, 2004.

[69] F. Piroi and 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 and A. Voronkov. Vampire. In [35], pages 292–296, 1999.

[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*, pages 209–216, Timişoara, Romania, 2001.

[73] J. Robu. *Automated Geometric Theorem Proving.* PhD thesis, RISC, Johannes Kepler University, Linz, Austria, 2002.

[74] E. Rodriguez-Carbonell and D. Kapur. Automatic generation of polynomial loop invariants: Algebraic foundations. In *Proc. of the Int. Symposium on Symbolic and Algebraic Computation, ISSAC'04*, pages 266–273, Santander, Spain, 2004. ACM Press.

[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], pages 1–8, 2004. Invited talk.

[77] M. Rosenkranz. A new symbolic method for solving linear two-point boundary value problems on the level of operators. *J. Symbolic Computation*, 39(2):171–199, 2005.

[78] M. Rosenkranz, B. Buchberger, and 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 Communications*, 15(2/3):111–126, 2002.

[81] J. Siekmann and C. Benzmüller. Omega: Computer supported mathematics. In S. Biundo, T. Frühwirth, and G. Palm, editors, *Proc. of the 27th German Conf. on Artificial Intelligence, KI'04*, volume 3238 of *LNCS*, pages 3–28, Ulm, Germany, 2004. Springer.

[82] J. Siekmann, S. Hess, C. Benzmüller, L. Cheikhrouhou, A. Fiedler, H. Horacek, M. Kohlhase, K. Konrad, A. Meier, E. Melis, M. Pollet, and V. Sorge. LOUI: Lovely Omega User Interface. *Formal Aspects of Computing*, 11(3):326–342, 1999.

[83] D. R. Smith. Mechanizing the development of software. In M. Broy and R. Steinbrueggen, editors, *Calculational System Design, Proc. of the NATO Advanced Study Institute*, pages 251–292, Amsterdam, 1999. IOS Press.

[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 Journal of Combinatorics*, 1(2):175–188, 1980.

[86] G. Sutcliffe and C. Suttner. The TPTP problem library: CNF Release v1.2.1. *J. Automated Reasoning*, 21(2):177–203, 1998.

[87] T. Tammet. Gandalf. *J. Automated Reasoning*, 18(2):199–204, 1997.

[88] A. Trybulec and H. A. Blair. Computer aider reasoning. In R. Parikh, editor, *Proc. of the Conf. Logic of Programs*, volume 193 of *LNCS*, pages 406–412. Springer, 1985.

[89] D. H. D. Warren. Higher-order extensions to Prolog: are they needed? In *Machine Intelligence*, volume 10, pages 441–454. Edinburgh University Press, Edinburgh, UK, 1982.

[90] C. Weidenbach, B. Afshordel, U. Brahm, C. Cohrs, T. Engel, E. Keen, C. Theobalt, and D. Topic. System abstract: Spass version 1.0.0. In [35], pages 378–382, 1999.

[91] F. Wiedijk. Comparing mathematical provers. In A. Asperti, B. Buchberger, and J. H. Davenport, editors, *Proc. of the Second Int. Conf. on Mathematical Knowledge Management*, pages 188–202, Bertinoro, Italy, 2003. Springer.

[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*. Wolfram Media Inc., 5th edition, 2003.

[94] W. T. Wu. Basic principles of mechanical theorem proving in elementary geometries. *J. Automated Reasoning*, 2:221–252, 1986.