# Theorema 2.0: Computer-Assisted Natural-Style Mathematics

BRUNO BUCHBERGER, TUDOR JEBELEAN, TEMUR KUTSIA,
ALEXANDER MALETZKY[1], and WOLFGANG WINDSTEIGER
Research Institute for Symbolic Computation (RISC), University of Linz (JKU), Austria

The *Theorema project* aims at the development of a computer assistant for the working mathematician. Support should be given throughout all phases of mathematical activity, from introducing new mathematical concepts by definitions or axioms, through first (computational) experiments, the formulation of theorems, their justification by an exact proof, the application of a theorem as an algorithm, to the dissemination of the results in form of a mathematical publication, the build up of bigger libraries of certified mathematical content and the like. This ambitious project is exactly along the lines of the QED manifesto issued in 1994 (see e.g. `http://www.cs.ru.nl/~freek/qed/qed.html`) and it was initiated in the mid-1990s by Bruno Buchberger. The *Theorema system* is a computer implementation of the ideas behind the Theorema project. One focus lies on the *natural style* of system *input* (in form of definitions, theorems, algorithms, etc.), system *output* (mainly in form of mathematical proofs) and *user interaction*. Another focus is *theory exploration*, i.e. the development of large consistent mathematical theories in a formal frame, in contrast to just proving single isolated theorems. When using the Theorema system, a user should not have to follow a certain style of mathematics enforced by the system (e.g. basing all of mathematics on set theory or certain variants of type theory), rather should the system support the user in her preferred flavor of doing math. The new implementation of the system, which we refer to as *Theorema 2.0*, is open-source and available through GitHub.

## 1. AN INTRODUCTION TO THEOREMA

The first author initiated the Theorema project around 1995, see for example [Buc97], after many years of experimenting with using formal predicate logic as a working language both in mathematical research as well as in mathematical education, see our early book on Mathematics for Computer Science [BL79], in which we taught predicate logic (for first semester students) as a working language for formulating problem specifications, algorithms, correctness theorems for algorithms and their proofs. So, when QED was first issued, we welcomed this as a fascinating vision and goal and, at that time, we had the first sketch and experiments towards Theorema under way. In distinction to projects that aimed at the implementation of efficient and general automated reasoners for proving isolated theorems, in the Theorema project we emphasized from its outset that the system should be a logic and software tool for supporting the entire process of mathematical theory exploration, i.e. definitions of notions, formulation and proof of propositions, formulation of problems, formulation and execution of algorithms for solving problems; all this with a strong emphasis on a structured and layered build-up of theories. We expressed this view continuously in the frame of the early EU Calculemus Project of which Theorema was one of the initiators, see for example [BMTV97].

[1] Supported by the Austrian Science Fund (FWF): grant no. W1214-N15, project DK1.

After a couple of years of experimenting with various programming languages, computer algebra and automated reasoning systems, in 1995, we had come to the conclusion that all the ingredients (attractive personal computers, screens, the internet, software technology, computer algebra libraries, attractive man-machine interfaces, two-dimensional syntax etc.) were available for designing and implementing a practically attractive mathematical theory exploration system. Based on a systematic comparison of systems available, we decided to use Mathematica as a frame for the development. The three reasons for this decision were: First and most importantly, the Mathematica programming language with its pattern matching programming style makes it easy to program reasoners in a straightforward and natural way. Second, Mathematica at that time was the only mathematical software system that already had an attractive user-interface (two-dimensional programmable syntax, structured notebooks, fantastic graphics tools, etc.). Third, Mathematica comes with a huge library of efficiently coded numeric, algebraic, and symbolic algorithms. The third point must not be misunderstood: The logic basis of Theorema does not rely on any of the algorithms in the Mathematica library but, rather, only uses Mathematica as a programming language (as meta-language for the implementation of Theorema reasoners). Still, sometimes, it may speed up experiments in the exploration of more advanced areas of mathematics if one may use some of the algorithms in a library, as black boxes, for conjecture forming, comparison, and other heuristic purposes.

The main design principles of Theorema were, and still are:

— The system *should support all phases of mathematical theory exploration.*

— The system should allow to *build up mathematical theories in a structured way.* (Organizational tools for this come for free with Mathematica. Internally, in Theorema, we implemented powerful functor constructs that allow to build up towers of mathematical domains conveniently, see [Buc96].)

— The basic logic language in which all theories are formulated is *higher order predicate logic* with currying (i.e. we allow formulae, for example, of the form $D[+][x, y]$, i.e. function $D$ applied to object $+$ yields a function that can be applied to $x$ and $y$). The user may then decide how she wants to build up mathematics (the part of mathematics she is interested in). In particular, *she may decide which kind of typing* should be used. For example, some user may want to build up everything in the frame of Zermelo-Fraenkel set theory, i.e. within first-order predicate logic with the universal membership predicate. Others may want to build up towers of domains with our functors that include a means for describing types in a very explicit way that distinguishes between different data structures for isomorphic domains.

— The advanced user of the system will want to explore a mathematical theory by *working on the object- and the meta-level in parallel*: she may, in certain stages of the exploration, work on the object-level by adding a few definitions, theorems, algorithms and call the reasoner that is currently available for the theory for proving, solving, or computing formulae in the theory. Alternatingly, she may work on the meta-level by extending the available reasoner. (In certain stages, the user may want to stay a long time on the object-level and just work on the theory with the currently available reasoners.) In Theorema, the object language is predicate logic and the meta language, for implementing reasoners, is Mathematica.

— *Reasoning* in Theorema is considered to have three different aspects: *proving, solving, and simplifying* (in particular, computing, i.e. simplifying ground terms). In practical reasoning, the interaction of these three aspects is of utmost importance and interest.

— Correspondingly, *proving* (in particular proving the correctness of algorithms relative to formal problem specifications) and *computing* (i.e. the execution of algorithms on data) is possible in the same language. In a future version, also *solving* (i.e. finding objects that satisfy certain properties depending on given objects) will be supported in the language.

— The result of reasoning (using the reasoners available for a particular theory) are abstract reasoning objects, which can then be post-processed for delivering readable reasoning text (in particular, *human-readable proofs*). We emphasize the importance of readable proofs in contrast to only producing proof scripts.

— The system should have a couple of *organizational tools* that allow to organize libraries of theories, display and navigate through proofs, design two-dimensional syntax according to the taste of the user, etc.

We are aware that a considerable number of projects and systems with similar goals as Theorema are available or under construction, e.g. Isabelle/HOL [WPN08], Coq [BC04a], Mizar [MR05, TB85], HOL/Light [Har96] and its relatives from the HOL family, Minlog [BMSS11], ACL2 [KMM00], Leo [WSB14], Matita [ARSCT11], and many more. A systematic comparison has been made a couple of times in the past twenty years, see for example [Wie06]. An in-depth comparison between all these systems and the Theorema system is not the aim of this paper. We think that, while the goals and views are similar, there are a couple of aspects and features in which Theorema differs from one or the other system:

— We emphasize the importance of *special provers for special theories* in the same way as, in the frame of mathematical software systems, special solvers are made available for special domains, e.g. solvers for linear and non-linear systems, solvers over various coefficients domains, solvers for differential and functional equations etc.

— Theorema's main credo is *natural style* in many aspects and facets. System input written by the user should be as natural as possible for our main target audience, namely mathematicians. We therefore put a lot of effort into the design and styling of Theorema content notebooks and the Theorema user interface in general, see Section 2.1. Also the system's output should delight the user. In the case of proving, the resulting proof should not only be a correct proof, but it should be in natural style, easy to read and understand, and nicely formatted. It should be presented in such a way that it is easy to browse and navigate through. In this respect, we see a chance that computer-generated formal proofs could even be perceived by users as an improvement compared to traditional proof presentation.

— Theorema should *not only* be a *proving system*, but also a *computing system* that allows efficient development and execution of mathematical algorithms and a *universal solving environment* that supports the specification and solution of various kinds of mathematical problems. In particular, the Theorema language should be rich enough to serve as a logical frame for all these activities.

— Many mathematical assistant systems are *interactive theorem provers* or *proof checkers* (Isabelle, Coq, HOL, Mizar), which assist the user in writing correct proofs by either only allowing to formulate correct proof steps or checking each step in a proof provided by the user. The focus of Theorema, on the other hand, is on *fully or partially automated proof generation.*

— In the Theorema system, the term "interactive theorem proving" is used for an optional mode, in which the fully automated proof generation may be user-assisted, typically through hints for the prover given by the user. As an example, in interactive mode the proof search loop may ask the user for the next open proof situation to process instead of automatically choosing the left-most, see Section 3. Or, when proving an existential goal, the system may ask the user for a witness term in order to proceed. User guidance is employed in situations where current automated techniques are not powerful enough to perform the desired steps or, in educational use, full automation is too powerful and performs steps that we want the user to practice and understand.

— Also, we want to emphasize that the goal of Theorema, typically, is the support of building up theories like elementary and advanced analysis, Gröbner bases theory, the theory of functions and relations, number theory, graph theory, topology etc., which consist of a big number of definitions, propositions, algorithms and pertinent proofs, where each of the steps in the theory is of relatively small size and much of the mathematical intelligence lies in the build-up of the theory and in the intuition behind a few of the theorems. In this way, there are similarities, but also differences, to recent important projects like the verification of the Kepler Conjecture (led by T. Hales based mainly on HOL/Light with assistance from other systems, see [Hal12]) or the complete formalization of the Feit-Thompson Theorem in Coq (led by G. Gonthier, see [GAA$^+$13]).

In the frame of a first version of Theorema, some interesting formal mathematical research happened with a couple of surprising results:

— We had *a couple of reasoners for various areas of mathematics* implemented that yielded proofs in what we think is a quite attractive proof style that may convince working mathematicians about the value of formal theory exploration systems, see [BDJ$^+$00] for a summary.

— We developed new methods for reasoning in natural style in general predicate logic, using meta-variables [KJ01] as well as a special technique for formulae with alternating quantifiers ("S-decomposition" [JBK$^+$09]), which allows an efficient usage of algebraic techniques for discovering witnesses and instantiation terms [VJB09].

— We approached the problem of program verification by creating a simple theoretical framework for the generation of verification conditions [PJ11, EJ10], in which we successfully combined logical and algebraic techniques for the automatic generation of polynomial loop invariants [KJ06].

— We implemented a *general version of Buchberger's algorithm using our Theorema functors* so that the algorithm works in arbitrary towers of domains (so-called reduction rings). Also, we developed a *new algorithm synthesis algorithm* (called "lazy thinking method") by which it was possible to *synthesize Buchberger's algorithm automatically from a formal specification of the Gröbner bases construction problem*, see [Buc04, BC04b]. This was somehow amazing because the Gröbner construction problem is a problem that, before Buchberger's PhD thesis in 1965, was open for 65 years.

— We developed *the first completely general symbolic method for solving linear boundary value problems*, which is based on showing that an analogue to the Baxter axioms for the differentiation, integration, and evaluation operators form a Gröbner basis (in some quite abstract non-commutative polynomial ring). This proof was done completely automatically using a Theorema implementation of polynomial reduction, see [RRTB11].

— The Theorema system has been used to support the teaching of mathematics. In particular, special learning units have been developed with the aim of making the "art of proving" accessible to beginner students. Tools have been developed, for instance, that allow the student to investigate how the choice of the appropriate knowledge base influences the proof that can be given, see [MSW07].

For the progress made in the Theorema project over the years, we refer to the overview papers [BJK$^+$97, BDJ$^+$00, BCJ$^+$06, JBK$^+$09]. The system grew both in size and capabilities through numerous contributions of both senior group members and PhD students. Many of the students, after finishing their theses, continued their careers in other places, and even changed their focus of research. On the side of the software, this posed certain challenges to the coherent maintenance of heterogeneous pieces of code. Together with advances in the user interface capabilities of the underlying Mathematica system, this made a re-design and re-implementation of the entire system necessary. The new version, which we call Theorema 2.0, still keeps all the basic design principles described above, but uses the lessons we have learned from the first implementation of the system.

The goal of the current paper is twofold: First, we want to describe the main new design features and software details of Theorema 2.0 (see Sections 2 and 3) and also a recent improvement of the unification algorithm, which is one of the basic logical instruments for provers in Theorema (see Section 4). Second (see Sections 5 and 6) we want to give two case studies of theory explorations in the Theorema style with an emphasis on showing how the explorations of theorems and algorithms interact and how the object-level of formulating theories and the meta-level of formulating special inference rules go hand in hand.

## 2. THEOREMA 2.0

*Theorema 2.0* is the latest implementation of the Theorema system. In Section 1 we described the reasons for implementing the first version of Theorema in 1995 on the basis of Mathematica. The main points are, in our view, still valid. Therefore, also Theorema 2.0 is implemented in the Mathematica programming language (since recently called the "Wolfram Language") and it uses the Mathematica front end as its user interface. In the following sections, we want to present the new system from two perspectives,

(1) *for users* who want to use the system as it is and develop their mathematical theories and

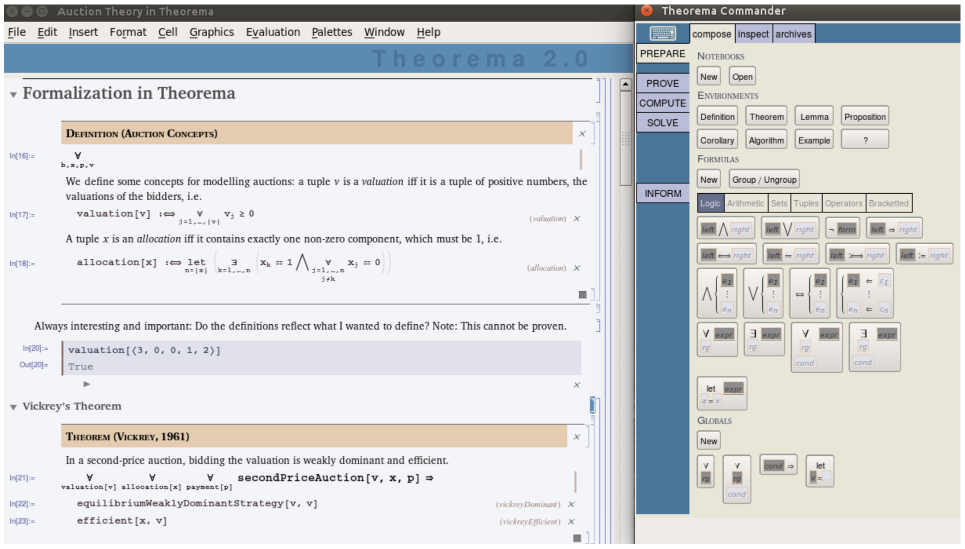
(2) *for developers* who want to enhance the system capabilities by adding new features.

Ideally, what we envisage as future users are expert users who combine the two views just described. Imagine they want to develop a certain field of mathematics, e.g. the theory of Hilbert spaces. Then, in addition to defining the needed concepts on the object-level, they provide their expert knowledge in the domain on the meta-level in form of inference rules for the newly introduced concepts. In the concrete example, they would formulate typical proof rules for the inner product in a Hilbert space. This reflects common practice in the hierarchical development of mathematics, where "advanced areas" are characterized not only by advanced content (object-level) but also by advanced techniques (meta-level). Providing meta-level inference definitions as part of the Theorema language on the user-level would require a mechanism of reflection with quoting, which has been studied as a prototype in Theorema 1, see [GB07]. We decided not to include such features in the current version mainly because our analysis in the technical report [GB07] was not thorough enough for making it possible to move from inference rules formulated and proved in Theorema to using the rules when exploring theories. We want to emphasize, however, that the perspective of formulating the reasoning rules within the Theorema language with the possibility of *proving their correctness within the system* is very attractive. It would certainly be a key feature of an envisaged QED-system, so we might pick up these ideas in a later stage of development.

## 2.1 The Theorema User Interface

We have a rather broad understanding of what a "user interface of a QED-system" must be capable of. We think of the user interface not only as a software component that allows the user to enter mathematical formulae or to call an automated prover. Rather, we want to model the whole process of doing mathematics and have it supported by the software as smoothly as possible. This concerns, in particular, the writing of mathematical documents. It is important to realize that typical mathematical documents (research papers, mathematical presentations, lecture notes, etc.) do not only consist of formal entities such as formulae, definitions, theorems, and proofs. In addition, they contain lots of semi- or informal parts such as text, graphics, organizational matters (e.g. declaring global variables for a part of the document), and the like. Recently, the notion of "flexiformal documents" has been introduced for this kind of material, see [Koh12].

In Theorema we address all these issues by using enhanced Mathematica notebooks as the primary format for writing all kinds of mathematical documents. Mathematica notebook documents consist of hierarchically nested cells, which all have a certain "style" that guides not only their appearance but also their behavior. Already in their standard version, they are kind of flexiformal, since there are different styles for input, output, text, typeset expressions, and many more. Of course, Mathematica notebooks are not perfect in all facets (when printed they are not as beautiful as a well-written LaTeX-document, they do not match BibTeX when it comes to literature citation, and when used as a beamer-presentation they do not support all features of a dedicated presentation software), *but* they are a


Auction Theory in Theorema
File Edit Insert Format Cell Graphics Evaluation Palettes Window Help
Theorema 2.0
Theorema Commander
compose inspect archives
PREPARE
PROVE
COMPUTE
SOLVE
INFORM
Notebooks
New Open
Environments
Definition Theorem Lemma Proposition
Corollary Algorithm Example ?
Formulas
New Group / Ungroup
Logic Arithmetic Sets Tuples Operators Bracketted
Globals
New

Formalization in Theorema

Definition (Auction Concepts)

In[16]:= $\forall_{b,x,p,v}$

We define some concepts for modelling auctions: a tuple *v* is a *valuation* iff it is a tuple of positive numbers, the valuations of the bidders, i.e.

In[17]:= valuation[v] :⟺ $\forall_{j=1,\ldots,|v|} v_j \geq 0$ (*valuation*)

A tuple *x* is an *allocation* iff it contains exactly one non-zero component, which must be 1, i.e.

In[18]:= allocation[x] :⟺ $\text{let}_{n=|x|}\left(\exists_{k=1,\ldots,n}\left(x_k == 1 \bigwedge \forall_{\substack{j=1,\ldots,n\\ j\neq k}} x_j == 0\right)\right)$ (*allocation*)

Always interesting and important: Do the definitions reflect what I wanted to define? Note: This cannot be proven.

```
In[20]:= valuation[⟨3, 0, 0, 1, 2⟩]
Out[20]= True
```

Vickrey's Theorem

Theorem (Vickrey, 1961)

In a second-price auction, bidding the valuation is weakly dominant and efficient.

In[21]:= $\forall_{\text{valuation}[v]}\ \forall_{\text{allocation}[x]}\ \forall_{\text{payment}[p]}$ secondPriceAuction[v, x, p] ⇒

In[22]:= equilibriumWeaklyDominantStrategy[v, v] (*vickreyDominant*)

In[23]:= efficient[x, v] (*vickreyEfficient*)

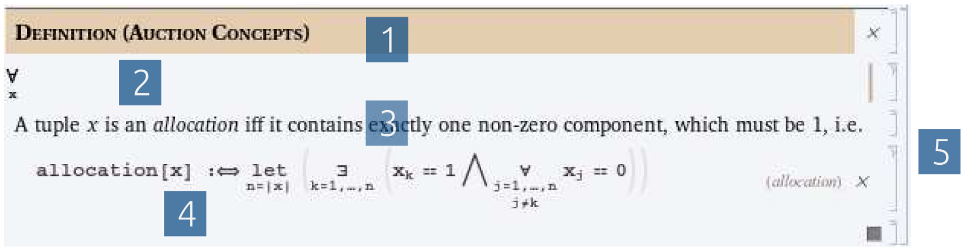
Fig. 1. Theorema 2.0 GUI: Content notebook left with Theorema commander to the right.

good compromise as a single format that can serve as the basis for many purposes. In general, we prefer to have only one format in which the user does her work compared to having many different tools with different formats and a variety of conversion mechanisms between all of them. Moreover, notebooks are interactive and can contain machine-executable parts, which we consider as very attractive. Last but not least, the notebook interface is a central component of the Mathematica system, and the chances are high that improvements can be seen with every new release of Mathematica.

Theorema notebooks can also contain big portions of system-generated content (formal output), most prominently automatically generated proofs. In order to initiate such content generation we provide a mouse-click driven user interface that is also responsible for supporting hand-crafted input of formal parts of the document (formal input). For Theorema 2.0 we have tried to take advantage of recent developments for supporting the implementation of user interfaces in Mathematica. The user interface of Theorema 2.0 now consists of two components, the *content notebooks* and the *Theorema commander*, see a screenshot of a Theorema session in Figure 1.

2.1.1 *Theorema Content Notebooks.* The content notebooks are enhanced Mathematica notebook documents. All typesetting facilities of Mathematica can be used in both formal and informal parts, and stylesheets (similar to cascading stylesheets in webpages) can be used to define the cells' appearance and behavior. Special cell styles are available for Theorema-specific formal portions of the document.[2] Most prominently, we provide so-called *Theorema environments*, which contain formal expressions to be processed by Theorema, typically definitions, the-

[2]Note that inside a cell it is not possible to mix formal with informal content, a cell is either entirely formal or entirely informal.



DEFINITION (AUCTION CONCEPTS)

$\forall_{x}$

A tuple $x$ is an *allocation* iff it contains exactly one non-zero component, which must be 1, i.e.

$$\text{allocation}[x] :\Longleftrightarrow \underset{n=|x|}{\text{let}} \left( \underset{k=1,\ldots,n}{\exists} \left( x_k = 1 \bigwedge \underset{\substack{j=1,\ldots,n \\ j\neq k}}{\forall} x_j = 0 \right) \right)$$ (allocation)

Fig. 2. Theorema environment expressing the definition of a new property

orems and the like. Figure 2 shows an example of a Theorema definition in detail. An environment starts with a header cell (marked (1) in Figure 2), which has no formal restrictions on its content but only serves as a structural element. It ends with a "■", which enforces automatic grouping of the cells inside the environment (5). The body of an environment may contain informal textual explanations (3), formulae expressed in the object-level syntax of the Theorema language (4), and what we call *globals* (2). The Theorema object-level syntax is essentially unchanged from its predecessor version; it supports a huge variety of common special characters and two-dimensional expressions, which results in expressions very close to usual mathematical standards, see Figure 2. Basic arithmetic expressions, set, tuple, and function notations have built-in support and new notions can be introduced via definitions.

We will describe two important novelties introduced in Theorema 2.0 in more detail, *hierarchical formula input* and *global quantifiers/conditions* (short: *globals*). Hierarchical formula input (formula tree input) addresses the problem of operator precedence in mathematical notations. As the simplest example, consider inputting a formula like $\underset{x}{\forall} P \Rightarrow Q$. Traditionally, this would be entered left-to-right by a sequence of keystrokes

$$\forall - \text{move below} - x - \text{move back to baseline} - P - \Rightarrow - Q$$

and depending on the precedence between $\forall$ and $\Rightarrow$, it would be interpreted as

$$\text{(a)} \quad \underset{x}{\forall} (P \Rightarrow Q) \qquad \text{or} \qquad \text{(b)} \quad (\underset{x}{\forall} P) \Rightarrow Q.$$

Due to the absence of a commonly agreed standard of precedences between all imaginable operators—and even if there was one, the average user would typically not memorize it—the desired grouping should be indicated by the user. The idea in the new Theorema user interface is to enter formula skeletons via mouse-click or keyboard-shortcuts, where subformulae conserve the grouping induced by the sequence in which they were entered. Correct grouping is enforced by automatic insertion of parentheses as shown in Figure 2 (4). We consider the parentheses as crucial, because they show *to the reader* what the author wanted to express. However, we show them almost invisible in order not to end up with parentheses-dominated LISP-reminiscent expressions. In addition, subformulae can easily be re-grouped manually, and there is a presentation mode, where parentheses vanish completely at the price of the formula becoming uneditable.

In the example, in order to obtain formula (a), instead of starting with the $\forall$-symbol, one would enter the $\forall$-*skeleton* resulting in $(\underset{\square}{\forall}\ \square)$ and proceed recursively in all subexpressions indicated by "$\square$". The first subexpression is a simple $x$, in the second we enter the $\Rightarrow$-skeleton resulting in $(\underset{x}{\forall}\ (\square \Rightarrow \square))$ followed by $P$ and finally $Q$. Using the Mathematica standard TAB-key to cycle through all $\square$-placeholders, this results in a sequence of keystrokes

$$\boxed{\forall} - x - \text{TAB} - \boxed{\Rightarrow} - P - \text{TAB} - Q, \tag{1}$$

where $\boxed{s}$ stands for mouse-click or keyboard-shortcut for skeleton $s$. Analogously, we get formula (b) by

$$\boxed{\Rightarrow} - \boxed{\forall} - x - \text{TAB} - P - \text{TAB} - Q. \tag{2}$$

Hence, we can get unambiguous and nice-looking input with reasonable effort, in the example even without extra keystrokes. We want to emphasize that the proposed way of entering formulae requires understanding the tree-structure of expressions, since the difference between (1) and (2) just reflects the different trees representing formulae (a) and (b). However, this should not be a hurdle, because even novice users must understand the logical structure of the expressions they want to work with.

Secondly, Theorema 2.0 introduces *globals*, which should be thought of as a formalization of the common "Let $K$ be a field ..." at the beginning of a section or the "Let $f$ be a function from $A$ to $B$ ..." at the beginning of a definition or a theorem. Both of these phrases are used to express that *in all what follows* certain symbols are used as universally quantified variables, possibly with conditions attached to them. It reflects common mathematical practice to either entirely omit the outermost universal quantifiers and interpret free variables as implicitly universally quantified or indicate universally quantified variables and conditions attached to them in the surrounding informal text. Logically, however, there is no difference to writing the universal quantifier with appropriate conditions in each formula. It is exactly this, what we want to bring to a formal level through globals. They are nothing more than particular formula fragments that are used to augment expressions within their scope. The scope of a global ranges over groups of formulae that are determined by the hierarchical grouping of cells in the notebook. Globals are written in a separate cell, see Figure 2 (2), and they affect all formulae that appear *after* the globals-cell in the nearest enclosing cell group. Notebook cells are grouped in sections, sub-, subsubsections, and finally environments, thus a global always applies to all formulae until the end of the current sectional unit or environment.

Theorema currently supports three kinds of globals:

*the global-*$\forall$ (written as $\underset{\substack{R\\C}}{\forall}$ with variable range $R$ and condition $C$) simply prefixes all formulae in its scope with the corresponding object-level $\underset{\substack{R\\C}}{\forall}$,

the *global-*$\Rightarrow$ (written as $C \Rightarrow$) turns every formula $F$ in its scope into $C \Rightarrow F$, and the *global-let* (written as $\underset{a=e}{\text{let}}$) defines an abbreviation $a$ for expression $e$ that is immediately expanded as soon as a formula in its scope is evaluated.[3]

*Example* 1. Given a global-cell on the section-level containing

$$\underset{x,y}{\forall}\ \underset{n=|x|}{\text{let}}\ |y| = n \Rightarrow$$

and assume $F_1, \ldots, F_m$ are the formulae in the remainder of that section, then what Theorema finally sees are the formulae

$$\underset{x,y}{\forall}\ |y| = |x| \Rightarrow F_1, \ldots,\ \underset{x,y}{\forall}\ |y| = |x| \Rightarrow F_m,$$

and quantifiers are only added for those variables that actually appear free in $F_i$.

*Example* 2. Suppose we are in the process of exploring the topic "functions from a set $A$ to a set $B$" and we already have defined earlier a predicate "$f$ is a function from $A$ to $B$" (denoted by $f : A \to B$). If we then want to introduce the basic concepts injectivity and surjectivity, one would typically write something like

Given sets $A$ and $B$ and $f : A \to B$, we define:

$$f \text{ is injective from } A \text{ to } B :\Longleftrightarrow \underset{x,y\in A}{\forall} f(x) = f(y) \Rightarrow x = y$$

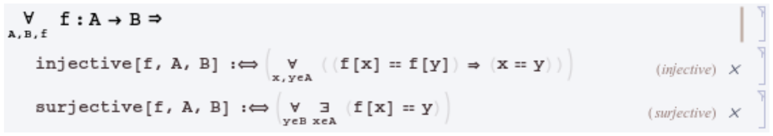
$$f \text{ is surjective from } A \text{ to } B :\Longleftrightarrow \underset{y\in B}{\forall}\ \underset{x\in A}{\exists} f(x) = y$$

In a formalized setting, this would translate into something like

$$\underset{A,B,f}{\forall} f : A \to B \Rightarrow (\text{injective}(f, A, B) :\Longleftrightarrow \underset{x,y\in A}{\forall} f(x) = f(y) \Rightarrow x = y)$$

$$\underset{A,B,f}{\forall} f : A \to B \Rightarrow (\text{surjective}(f, A, B) :\Longleftrightarrow \underset{y\in B}{\forall}\ \underset{x\in A}{\exists} f(x) = y).$$

Not only does this need more typing, it is also more complicated to read and comprehend as the informal representation above. Using globals in Theorema 2.0 the same can now be expressed compactly inside a definition-environment as follows:



$\underset{A,B,f}{\forall}$ f : A → B ⇒

injective[f, A, B] :⇔ ($\underset{x,y\in A}{\forall}$ ((f[x] == f[y]) ⇒ (x == y))) (*injective*) ×

surjective[f, A, B] :⇔ ($\underset{y\in B}{\forall}\ \underset{x\in A}{\exists}$ (f[x] == y)) (*surjective*) ×

Similarly, globals can be used to introduce section-wide global variables with appropriate conditions valid in certain parts of a document, which is an important aspect in Theorema's ambition to make formal mathematics appear close to traditional style.

2.1.2 *The Theorema Commander.* The Theorema commander, see Figure 1, is the component in the user interface that supports the creation of content notebooks, in particular those parts that can be generated with automation-support. It has been introduced in [Win12], and its main purpose is to support the main

[3]In mathematics as well as in programming languages the phrase "let" occurs in various situations. In mathematics, it often expresses universal quantification. It is important not to confuse the global-$\forall$ and the global-let.

*activities* proving, computing, and solving[4] and also some administrative aspects like document preparation (e.g. support for two-dimensional formula input) and session organization. For each activity selected on the left margin of the commander, there are several concrete *actions* to be performed in order to complete the activity. Actions are selected through menu buttons in the top row of the main window, and the intention is that they are run through left-to-right. All actions are carried out in point-and-click interfaces and result in the appropriate configuration of the respective activity. The result of the activity is then integrated as formal output into the content notebook, from which the activity was initiated.

*Example* 3. In order to get a proof generated automatically in Theorema 2.0, the user must

—specify the proof goal,

—specify the knowledge base available in the proof

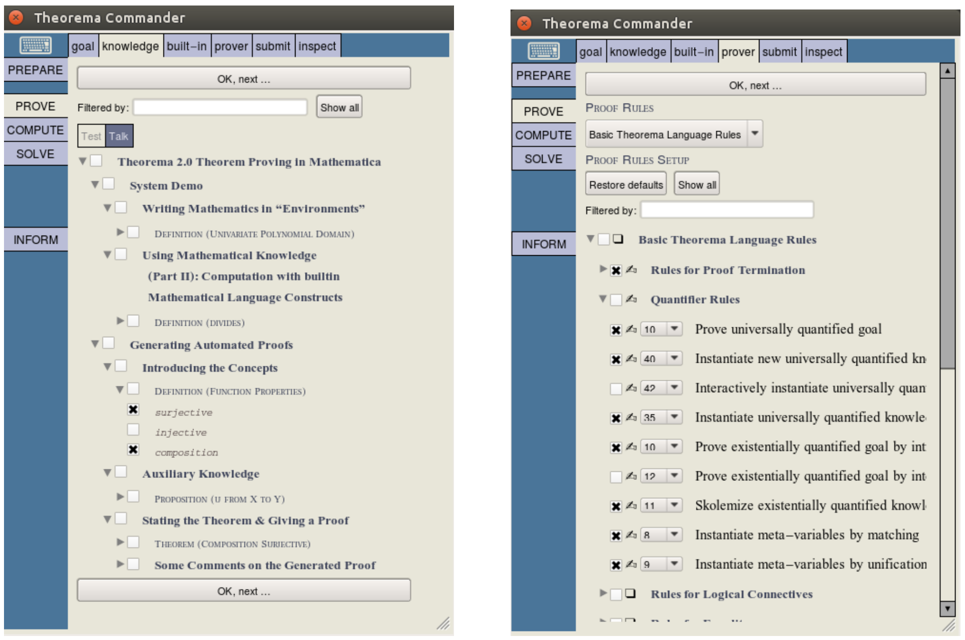
—specify available knowledge about Theorema built-in language constructs, and

—select and configure the desired proof method.

Correspondingly, the 'prove'-activity consists of separate actions guiding the user through these steps. In the 'goal'-action, the proof goal is defined by just selecting the formula to be proved in the notebook. In order to quickly compose the knowledge base, the 'knowledge'-action displays the *knowledge browser*, which consists of a summary of formulae for each open notebook, hierarchically structured according to the grouping given in the notebook, see the illustration in Figure 3. For each formula only its textual label is displayed, the full formula is unveiled in form of a tooltip as soon as the mouse moves over the label. By clicking the corresponding check-box, the formula (or an entire group) is selected to go into the knowledge base for the next proof. It serves for the orientation and navigation through the knowledge browser when telling text is used in the informal environment header cells and the formula labels (see Figure 2 (1) in Section 2.1.1). In a similar fashion, the user can select the computational knowledge about built-in language constructs.

A Theorema prover typically consists of inference rules that are applied using a certain proof strategy, see more details in Section 3. The 'prover'-action allows to (de)activate rules on an individual basis, the rule implementation defines rule groups that can again be (de)activated similar to the knowledge browser above. As soon as the prover configuration is finished, the 'submit'-action displays a compact summary of all settings chosen in the previous steps. The data can now be sent to the prover, which then tries to generate a proof. The commander steps to the 'inspect'-action, which features a live animation of the tree corresponding to the ongoing proof search. Tree nodes exhibit different shapes and colors depending on their type (AND-node/OR-node) and status (success/failure/pending). Proof generation stops as soon as one successful proof has been found or the search has reached its time or space limits. In the content notebook, we then find a link to the proof and information on proof generation configuration, which is very useful when the proof needs to be re-generated. When clicking the link, a natural language proof

[4]Support for general "solving" is not yet studied in detail and, hence, not yet implemented in the current release of the Theorema system, see also Section 5.



Theorema 2.0 Theorem Proving in Mathematica
System Demo
Writing Mathematics in "Environments"
Definition (Univariate Polynomial Domain)
Using Mathematical Knowledge (Part II): Computation with builtin Mathematical Language Constructs
Definition (divides)
Generating Automated Proofs
Introducing the Concepts
Definition (Function Properties)
☒ surjective
☐ injective
☒ composition
Auxiliary Knowledge
Proposition (u from X to Y)
Stating the Theorem & Giving a Proof
Theorem (Composition Surjective)
Some Comments on the Generated Proof

Proof Rules
Basic Theorema Language Rules
Proof Rules Setup
Basic Theorema Language Rules
☒ Rules for Proof Termination
☐ Quantifier Rules
☒ 10 Prove universally quantified goal
☒ 40 Instantiate new universally quantified kn
☐ 42 Interactively instantiate universally quan
☒ 35 Instantiate universally quantified knowle
☒ 10 Prove existentially quantified goal by int
☐ 12 Prove existentially quantified goal by int
☒ 11 Skolemize existentially quantified knowl
☒ 8 Instantiate meta–variables by matching
☒ 9 Instantiate meta–variables by unification
☐ Rules for Logical Connectives

Fig. 3. The knowledge browser (left) and the prover configuration (right)

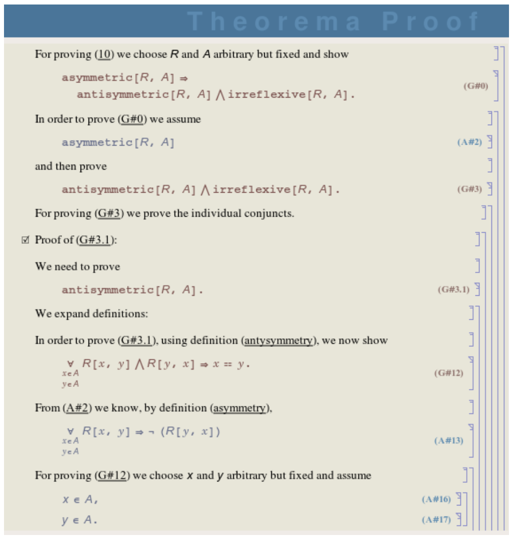
presentation generated automatically from the underlying proof object is displayed in a separate notebook, similar to how proofs were displayed in Theorema 1. Note that when we speak about proofs or proof objects, we always mean "possibly partial proofs / proof objects". Theorema always generates a proof object, even if the proof fails or is aborted. A natural language presentation can be generated for a failing or incomplete proof attempt as well. The status of a proof can be read off from the status of the root node in the tree presentation (currently green with a ✓-indicator, though this may become configurable in the future) and from a small success indicator icon displayed in the content notebook next to the link to the proof.

The proof notebook and the proof tree, see Figure 4 left and right, are bidirectionally linked:

—selecting a cell in the proof notebook marks the corresponding node in the tree;
—clicking a node in the tree moves the cursor to the corresponding step in the proof notebook.

Furthermore, we make heavy use of tooltips, which allow to display additional information when the mouse moves over certain objects, e.g. formula labels in a proof carry the whole formula in a tooltip, and nodes in the proof tree carry information on the inference rule that was applied at that point.

We are confident, and first observations with novice users support this claim, that the new interface makes working with Theorema 2.0 a lot easier compared to its predecessor, in particular for non-expert users. Furthermore, the novel proof navigation provides much better understanding of the proof structure.



Theorema Proof

For proving (10) we choose *R* and *A* arbitrary but fixed and show

```
asymmetric[R, A] ⇒
  antisymmetric[R, A] ⋀ irreflexive[R, A].                (G#0)
```

In order to prove (G#0) we assume

```
asymmetric[R, A]                                            (A#2)
```

and then prove

```
antisymmetric[R, A] ⋀ irreflexive[R, A].                    (G#3)
```

For proving (G#3) we prove the individual conjuncts.

☑ Proof of (G#3.1):

We need to prove

```
antisymmetric[R, A].                                      (G#3.1)
```

We expand definitions:

In order to prove (G#3.1), using definition (antysymmetry), we now show

$\forall_{\substack{x \in A \\ y \in A}} R[x, y] \wedge R[y, x] \Rightarrow x = y.$ (G#12)

From (A#2) we know, by definition (asymmetry),

$\forall_{\substack{x \in A \\ y \in A}} R[x, y] \Rightarrow \neg (R[y, x])$ (A#13)

For proving (G#12) we choose *x* and *y* arbitrary but fixed and assume

$x \in A,$ (A#16)

$y \in A.$ (A#17)

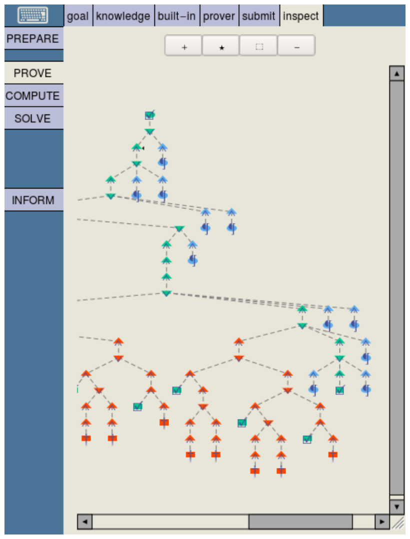


Fig. 4. The natural language proof presentation (left) and the corresponding proof tree (right)

## 2.2 Software Technological Aspects of Theorema 2.0

2.2.1 *Proving and Computing in One Language.* As already described in Section 1, we use Mathematica as the meta-language for the implementation of Theorema reasoners. At the same time, we use the Mathematica notebook front-end as the user interface and we want to provide the full power of Mathematica to all users even when the Theorema system is loaded, e.g. a user should still be able to solve a system of equations using the Mathematica `Solve`-command. On the other hand, the Mathematica language also subsumes mathematical expressions and it uses a particular implicit semantics, which we do not want to be the semantics of Theorema automatically, e.g. the syntactical expression `a+b` is represented internally as `Plus[a,b]`, which represents an operation `Plus` applied to two objects `a` and `b`. Mathematica, however, treats `Plus` always associative and commutative with 0 as identity. Furthermore, in the presence of concrete numeric values for `a` and `b` it might perform simplifications based on Mathematica built-in black-box knowledge.

Since Theorema is not based on Mathematica semantics, it is of utmost importance to strictly separate Theorema expressions from meaningful Mathematica expressions. The problem can, in principle, be solved easily, since using the `MakeExpression`-mechanism in Mathematica allows to redefine the entire Mathematica parser, such that e.g. `a+b` is no longer interpreted as `Plus[a,b]` but as `foo[a,b]`. This would, however, require huge effort to re-implement the entire Mathematica parser for all kinds of expressions. The solution in Theorema 1 was therefore to just rename all operators that carry a meaning in Mathematica once and for all to fresh symbols, see [Win01]. This turned out to be unstable when new Mathematica releases come out, in which previously unknown operators may suddenly have semantics attached. Errors of that kind are very subtle and were difficult to detect and isolate because they could just emerge as an undesired transformation of an expression deep inside the execution of one of our reasoners without further notice.

Thus, in Theorema 2.0 the solution to this problem is to rename *all symbols* to fresh symbols when they appear in a Theorema expression. We use the new possibility introduced in recent versions of Mathematica to define the behavior of notebook cells via the stylesheet, i.e. we introduce special cell types for writing Theorema expressions such that special expression preprocessing can be employed for these cells.[5] Two such cell types are those for formulae inside an environment and those for globals. These two differ in turn, because we want to interpret a '$\forall$' differently depending on whether it is written as a global or as part of a regular formula.

When it comes to computation, we want to provide semantics for a certain computable fragment of the Theorema language, namely integer numbers, rational numbers, (finite) tuples (i.e. lists), finite sets, boolean expressions, and quantifiers with finite ranges. These datatypes are built into Theorema, and the standard operations on them (arithmetic on numbers, standard tuple operations, set operations, etc.) are provided as part of the Theorema system, i.e. they are programmed in Mathematica and some of them use available Mathematica functionality. Most prominently, of course, arithmetic on integers and rational numbers uses available Mathematica algorithms, and tuples are represented as Mathematica list such that efficient list algorithms can be re-used. Sets, on the other hand, are not a basic datastructure in the Mathematica language, hence Theorema introduces its own representation based on Mathematica-lists, such that powerful algorithms for e.g. set intersection/union from Mathematica can be used. Special precautions are taken due to the fact that computation might interact with proving in the sense that a computation can appear as a simplification step during a proof and, hence, expressions may contain variables.

*Example* 4. For finite sets, their intersection just contains the finitely many objects in common, e.g. $\{1,5\}\cap\{3,5\}$ gives "by computation" $\{5\}$. On the other hand, $\{1,5,a\}\cap\{3,5\}$ (with a variable $a$) should *not* compute $\{5\}$, because the result is either $\{5\}$ or $\{3,5\}$ depending on the value of $a$. Incautious computation could then lead to erroneous proof steps, e.g. when we are to prove $\underset{a\in A}{\exists} |\{1,5,a\}\cap\{3,5\}| = 2$.

Computations are driven by the standard rewriting mechanism of Mathematica, which is based essentially on substitution and replacement like in predicate logic. This means, that Theorema semantics must be represented on the level of Mathematica function definitions, which are then applied as rewrite rules by the Mathematica computation engine. This puts Theorema semantics functions to the same level as Mathematica built-in functions with the problem of undesired evaluation inside of formulae as described at the beginning of this section. The key to a solution is given by *Mathematica contexts*, which provide appropriate *namespaces* for Mathematica symbols. Let us consider the simplest example `1+1` as part of a Theorema formula. The internal full form is `Plus[1,1]`, which we rename to a "fresh" `Plus$TM[1,1]` in order to avoid confusion with any semantics for `Plus`. Furthermore, when entered in a formula inside an environment, the formula pre-

[5]This explains the restriction mentioned in Section 2.1.1 that formal and informal content cannot be mixed inside one cell.

processor associated with that cell type sets up the parser in such a way that the expression actually becomes

```
Theorema`Language`Plus$TM[1,1]
```

where `Theorema`Language`` is the designated context for symbols known in the Theorema language. Everything related to the built-in computational semantics has its home in `Theorema`Computation`Language``, i.e. the definition of Theorema's built-in + for numbers defines `Theorema`Computation`Language`Plus$TM`, and it is guaranteed that all semantic-related definitions are given for symbols in the computation-context *only*.

For doing computations, Theorema notebooks provide special computation cells, in which users must write their input. In that situation, the formula preprocessor turns `1+1` into

```
Theorema`Computation`Language`Plus$TM[1,1]
```

which will immediately simplify as expected to `2` by built-in Theorema semantics. For computations as part of a proof, the computation must be initiated explicitly by the prover programmer. For this purpose, the prover programming library provides translation functions between the Theorema language context and the corresponding computation context.

When a user enters the definition of a function or a predicate inside an environment, this should always be available for user computations also. Consider as an example the definition

$$\underset{v}{\forall} \quad \text{valuation}[v] :\Leftrightarrow \underset{j\in 1,\ldots,|v|}{\forall} v_j \geq 0$$

from Figure 1 in Section 2.1. In order to make this definition accessible inside a Theorema computation, it is translated in the background automatically to a Mathematica function definition, which is then evaluated in the computation context such that it is immediately available for the rewrite engine in subsequent computations. In addition, a condition is attached, that allows to (de)activate the definition from within the user interface, so the final definition looks like

```
Theorema`Computation`Knowledge`valuation$TM[v_] /; active[...] :=
   Theorema`Computation`Language`Forall$TM[...]
```

where `active[...]` checks a flag connected to the check-box at the respective definition in the knowledge browser of the 'compute'-activity[6] and `Forall$TM[...]` stands for the Theorema datastructure representing the ∀-formula on the right-hand side of the definition. Due to the semantics of the `Forall$TM` that is present in the computation context, an expression valuation[$\langle 3, 0, 0, 1, 2\rangle$] in a computation cell will immediately rewrite to *True*, see again Figure 1. We should mention that also all Theorema built-in functions carry a similar condition in their definitions such that they can be (de)activated via the user interface. For this, both the 'prove'- and the 'compute'-activity have a 'built-in'-action that features check-boxes for

[6]Figure 3 shows the knowledge browser in the 'prove'-activity, where checking a box has the effect of putting a formula in the knowledge base of the proof. The knowledge browser in the 'compute'-activity looks exactly the same, but checking a box there means to set the corresponding flag.

all built-in operations and works essentially like the 'knowledge'-action described in Section 2.1. If a user, for instance, decides that she does *not* want to rely on Theorema built-in semantics for $\forall$ then she would only need to deactivate $\forall$ in the GUI and a computation of valuation[$\langle 3,0,0,1,2\rangle$] would then return

$$\underset{j\in 1,\ldots,5}{\forall} \langle 3,0,0,1,2\rangle_j \geq 0.$$

2.2.2 *Community Support Through Knowledge Archives.* Knowledge archives serve the purpose of distributing mathematical theories among Theorema users. This is very important in practice, because otherwise every user would have to start building up mathematics based on just the elementary objects supplied by Theorema, i.e. tuples and sets. The main challenges that must be met are the following:

*Namespaces.* The names of symbols in one archive should not accidentally collide with symbol names in other archives or user-defined symbols. However, archives should support theory hierarchies, where theories depend on each other and symbols from one theory can safely be used inside an other.

*Ease of use.* It should be easy for the user to load a theory into the current Theorema session.

*Communication.* Theory archives should be exchangeable with other systems. This requires theories to be stored in some standard used also by other mathematical assistant systems, e.g. MathML, OpenMath, OmDoc, etc. Last but not least, archives should not depend on the platform they have been developed on.

These problems are well-known in the field of algorithm libraries for programming languages. For this case, Mathematica supports a mechanism for writing algorithm *packages*, which has proven over many years to satisfy the first two of the above criteria. The organization of Theorema archives is therefore oriented on the principles of Mathematica packages, which are based again on the concept of Mathematica contexts, see also Section 2.2.1. Theorema archives are composed in Theorema notebooks that use an archive header section at the beginning in order to setup the archive. It consists of the archive name (as a context $\texttt{dir}_1$`...`$\texttt{dir}_n$`name`), where `name` should be the basename of the notebook's filename and the $\texttt{dir}_1$ to $\texttt{dir}_n$ are the directories on the filename path (relative to the standard location of archives in Theorema), e.g. an archive for equivalence relations could have the name `algebra‘relations‘equivalence‘`, it must then be written in a notebook "equivalence.nb" in the subdirectory "algebra/relations" of Theorema's standard archive directory. Next in the header is a list of parent archives, on which the new archive depends. This has the effect that the new archive may use all symbols that are *exported* from its parents. Finally, there is a list of all symbols that the new archive wants to export (i.e. allow to be used outside) in addition to the exported symbols of its parents. The convention is that all symbols not explicitly exported are treated as private to the archive and hence invisible outside. The rest of the notebook is a plain Theorema content notebook, only at the very end it must have an indicator to close the archive.

When the archive is closed, all formulae written in environments in the notebook are stored as Mathematica expressions (representing Theorema formula datastructure) in an external file using Mathematica's standard file exporting features. In other words, a Theorema archive file contains just plain Mathematica expressions, hence loading an archive into a Theorema session can simply be done with the Mathematica <<-command well-known from loading standard Mathematica packages. It is important to note that we do not store any informal content in an archive, and we store the parsed and fully processed formulae and *not* the two-dimensional formula structures contained in the notebook. In particular, all globals appearing in the notebook (see Section 2.1.1) are already applied and thus need not be stored in the archive. For distribution among users it is optional to distribute the notebook, from which an archive was generated, together with the archive in case the informal parts should be communicated as well. We store some interface information into the archive in addition such that, for instance, a knowledge browser entry for the archive will be added to the commander even if the respective notebook is not present.

Theorema archives are Mathematica expression files, so they can be read on any platform regardless from where they have been generated. Communication with other systems will require translations from Theorema syntax to MathML, OpenMath, etc. As an example, the implementation of a mechanism for saving an archive in the MMT-format, see [RK13], is under way at the time of writing this paper.

## 3. HOW PROVERS ARE ORGANIZED IN THEOREMA 2.0

In this section, we will describe the organization of reasoners in the Theorema 2.0 system, which, as its overall aim, tries to allow as much configuration as possible for the user without the need of a special language. The concept of a *reasoner* in Theorema 2.0 differs substantially from how provers were organized in its predecessor version. We consider a Theorema reasoner to be just a plain sorted list of inference rules accompanied by a strategy for their application. As already indicated in Section 2, giving the user the possibility to define inference rules and strategies would require non-trivial extensions of the language and is thus currently not supported. Apart from that, the user should have as much flexibility as possible in setting up her own reasoners. From now on we want to focus on the special case of reasoners for proving, i.e. on *Theorema provers.*

### 3.1 The User's View

The setup of the prover is done by the user in the 'prover'-action as part of the 'prove'-activity in the Theorema user interface, see Figure 3 in Section 2.1.2. Firstly, the user may choose a certain collection of inference rules via a menu from a set of predefined collections that are provided by prover developers. The actual rules provided by the selected collection are then displayed in structured groups that are fixed by the prover developer. Each rule comes with two check-boxes and a pull-down-menu containing numeric priority values. The left check-box (☐) allows to entirely (de)activate the respective rule, while the right check-box (visualized as a hand using a pencil) allows to turn a rule silent. A silent rule will apply during a proof as usual, but its application will not be reflected in the natural language presentation of the proof.

In order to understand the consequences of the prover setup, the user only needs to understand the basic principle of how provers work. Theorema maintains a *proof* *object*, which serves two purposes:

(1) during proof generation, it stores all information necessary for guiding the proof search and
(2) when the search has completed, it contains the proof tree. In case of successful completion the tree makes up the formal proof, otherwise it represents a partial proof attempt, which may still be instructive for the user to study.

The Theorema proof object is organized as a tree structure with each node representing of a *proof situation.* In addition, each node carries some *proof information* needed for proof display and a *proof status* relevant for this situation. The tree is an AND-OR-tree, meaning that each non-terminal node can either be of type AND or of type OR. AND-nodes stand for proof situations that are proved as soon as *all* of their children are proved, they correspond to nodes in usual formal proof trees. OR-nodes require only that *at least one* child is successfully proved, they are used essentially to model alternatives for the proof search to continue, and they are a means to efficiently implement backtracking in case of failing branches in the proof search. Terminal nodes in the tree are either of type OPENPROOFSITUATION indicating a situation where the proof needs to be continued, or of type TERMINALPROOFSITUATION with an indication of success or failure where the respective branch of the tree ends. Type and status of proof nodes is used in the visualization of the proof tree, see Figure 4 in Section 2.1.2. The main proof-loop is pretty standard and straightforward.

— It starts with a proof object consisting of only the root node. It is of type OPENPROOFSITUATION with status "pending" containing the initial proof goal and the user supplied knowledge base coming from the 'goal'- and 'knowledge'-actions in the user interface, respectively.

— As long as the root node has status "pending" and neither the maximum search depth nor the maximum search time have been reached, it locates *all open proof situations* in the tree, *determines the next one to be processed*, and finally *applies the strategy with all active inference rules to this situation*, where the rules' activity comes from the prover setup in the 'prover'-action of the user interface.

— Both inference rules and strategies in general return a proof tree. In the simplest case, the strategy could only apply one rule. Then the tree returned is just the tree returned by that rule. Otherwise, the strategy generates an OR-node with the trees returned by the individual rules as its subtrees.

— The proof status of the leaves is then recursively propagated depending on the node types towards the root of the proof tree and the loop continues.

The prover setup influences the list of rules that are actually applied during each loop-run. Firstly, the structured rule groups in the user-chosen collection are *flattened*, then *filtered* according to user-defined activation, and finally *sorted* with respect to ascending priority. The resulting list of rules is then applied to the open proof situation using the standard Mathematica `ReplaceList`-command for transformation rules, which returns a list of *all possible results.* The order of the proof situations in that list corresponds to the priorities of the inference rules. Hence, the user influence on the resulting proof is two-fold: activation and deactivation of rules decides on the presence of certain branches in the tree and the selected priorities decide on their order.

*Example* 5. The prover configuration in Figure 3 chooses the rule collection "Basic Theorema Language Rules", which consists of groups "Rules for Proof Termination" (all of them active as indicated by the checked box in the group header), "Quantifier Rules" (not all of them active as indicated by the unchecked box in the group header), "Rules for Logical Connectives" (not all active), etc. In the group of quantifier rules, all rules but two are activated, the rule for universal quantifiers in the goal has priority 10, and the rule for instantiating universal knowledge has priority 40, meaning that universal quantifiers in the goal will be treated *before* universal quantifiers in the knowledge base. No rule is silent, since the hand-pencil-icon is present for every rule.

The selection of the *next open situation* to be processed is currently always the left-most, resulting in a depth-first-like proof search. We deviate from strictly searching depth-first by checking each new situation immediately for proof success. This means precisely that we do one level of breadth-first search with a restricted rule set (only those that allow to determine success) before we continue depth first. Moreover, we integrate possibilities for semi-automated search by allowing user interaction in the choice of the next proof situation to expand. Independent from this kind of interactivity on the level of the proof search, we mention that interactivity may also be implemented at the rule level, i.e. inference rules that, when applied, require some sort of dialog-driven user interaction. Rules of that kind are implemented for instance in the specialized prover described in Section 6. The proof search finishes with the proof object representing

| | |
|---|---|
| *a successful proof* | in case the root node has status "proved", |
| *a failing proof* | in case the root node has status "failed", or |
| *an incomplete proof* | in case the root node has status "pending". |

In case of a successful proof, all OR-nodes can be eliminated by replacing them with their necessarily existing successful subtree. The resulting tree then corresponds to a *formal proof.*

From every proof object we can generate a natural language proof presentation (see Figure 4 in Section 2.1.2) using proof text templates, which the prover implementation must provide for each inference rule. These templates use the proof information stored with each node, which contains all formulae needed and generated by that rule.

## 3.2 The Developer's View

For a prover programmer, implementing a new prover means to

—implement individual inference rules,

—provide proof text templates for each rule, and

—define the rule group structure of the new prover.

Inference rules are implemented as Mathematica rewrite rules `lhs :> rhs`, which transform the input proof situation `lhs` into a proof tree that is produced by the

Mathematica program `rhs`. Every rule has a *name* that corresponds to an entry in the prover setup in the user interface. As an example, take the rule for proving a universally quantified goal. The name for this rule is `forallGoal` and it is stored as `inferenceRule[ forallGoal]` essentially in the form

```
  inferenceRule[ forallGoal] =
    PRFSIT$[ g:FML$[ _, u:Forall$TM[ rng_, cond_, A_], __],
             k_List, id_, rest___?OptionQ] :> rhs
```

`PRFSIT$[g,k,...]` is the Theorema datastructure representing a proof situation consisting of goal `g` and knowledge base `k` (and optional data `...`). Typically, we use the Mathematica pattern matching mechanism in the specification of `g` and `k` in order to describe, for which situations the rule should be applicable. Note that Mathematica patterns may even contain arbitrary conditions, which allow to check additional properties involving optional data present in the proof situation (e.g. information about the proof progress). In the concrete example, the program `rhs` would need to generate new constants $x_1, \ldots, x_n$ for the variables contained in the quantifier range `rng` and produce new formulae $g'$ and $c'$ corresponding to the quantifier's body formula `A` and the condition `cond` with the variables substituted by the constants $x_1, \ldots, x_n$, respectively. From these ingredients, a new proof tree node has to be created. The new proof situation consists of goal $g'$ and knowledge $\mathtt{k} \cup \{c'\}$, and the proof information tells that formula `g` was used to generate new formulae $g'$ and $c'$, and the rule applied was `forallGoal`. The Theorema prover programmer library provides the necessary functions to analyze formulae, do substitutions, create new proof tree nodes, and all that.

The template for english proof text ("En") for this rule looks something like

```
  proofStepText[ forallGoal, "En", {{g_}}, {{g'_, c'__}}, ___]:=
    {textCell[ "For proving ", formulaReference[g], " we choose ",
               inlineTheoremaExpressionSeq[ v, "En"],
               " arbitrary but fixed and assume"],
      assumptionListCells[ {c'}, ",", "."],
      textCell[ "We have to show"],
      goalCell[ g', "."]
    };
```

In the natural language presentation this would result in proof text

For proving *label* we choose $x_1, \ldots, x_n$ arbitrary but fixed and assume

$$c'.$$

We have to show

$$g'.$$

where *label* is the short label of the formula `g`.

Finally, for defining inference rule groups, we need code of the form

```
registerRuleSet[ "Quantifier Rules", quantifierRules,
  {
```

```
  {forallGoal, True, True, 10},
  {forallKB, True, True, 40},
  ...
  }]

registerRuleSet[ "Basic Theorema Language Rules", basicTmaLangRules,
  {
  terminationRules,
  quantifierRules,
  connectiveRules,
  ...
  }]
```

which defines the group `quantifierRules` consisting of the rules `forallGoal` (default activation: True, default presence in natural language presentation: True, default priority: 10), `forallKB` with respective default values, and several more rules. Based on that, the group `basicTmaLangRules` is defined to consist of the subgroups `terminationRules`, `quantifierRules`, `connectiveRules`, etc. This rule grouping corresponds exactly to the structured display of rules in the 'prover'-action shown in Figure 3.

The notion of inference rule maybe needs some clarification: typically, the focus of Theorema lies on inference rules representing one logical proof step, e.g. modus ponens or proof by contradiction. The natural language presentation is then capable of describing every logical step, we call these provers *white-box-provers*. Still, inference rules can also be implemented in such a way that a certain sequence of smaller steps is hardcoded within one rule, i.e. one bigger step. In other systems, this might be handled via tactics. Theorema uses dedicated rules instead, because making only one big step instead of many small ones also has the effect that only one step will be documented in the proof object. This is useful because then the textual proof explanation can describe the combined step appropriately, because the explanation of a combined step *is not* just the concatenated explanation of the individual steps. As a simple example, we could have one rule for proving universally quantified implications, which would essentially combine the rules for proving universal goals and proving implications but it would give the explanation in one stroke. For a nice application, we refer to Section 6.2. A more extreme example could be a *Gröbner basis prover* for proving boolean combinations of polynomial equalities and inequalities. Such formulae can be proved by the Gröbner basis method by transforming the formulae into the problem of the solvability of a system of polynomial equations, which can be decided by a Gröbner basis computation. The transformation process and the respective Gröbner basis computation could then be implemented altogether as *one rule*, and it is up to the prover programmer, how much information goes into the proof information in the proof object. We call such provers *black-box-provers*. In particular in such provers it might be convenient to have a powerful algorithm library like the entire Mathematica system available for prover programming. We want to stress, however, that white-box-provers typically only use Mathematica as a programming language. Even black-box-provers do not necessarily rely on built-in Mathematica algorithms, they could, for instance, also call external SAT or SMT solvers.

## 4. UNIFICATION IN THEOREMA

Unification is a key ingredient in theorem provers and proof assistants, used in a number of inference rules. It tries to make two terms identical or equal modulo an equational theory by replacing some variables by the corresponding expressions.

The core unification algorithm implemented in Theorema is unification modulo $\alpha$-equivalence, adapted to the special syntactic features of the system. This is a kind of pragmatic, minimalistic approach, since $\alpha$-equivalence can be seen as the most basic property of languages with binders. Developers of individual Theorema reasoners may choose to rely on the algorithm as it is, or to extend/modify it to meet the needs of that particular reasoner.

Note that although Theorema provides higher-order syntax, there is no hidden default higher-order logic behind it. Therefore, a developer of a special prover for higher-order logic may wish to extend the algorithm to deal with equalities modulo $\beta$ and $\eta$, while, e.g. for first-order reasoning, the provided algorithm would suffice.

The unification problem is formulated as a meta-equation between two Theorema expressions. The syntax is quite liberal: Such an expression may be a constant, a variable, an application of an expression to a sequence of expressions, or a quantified expression. Variables are of two kinds: for individual expressions (individual variables) and for sequences of expressions (sequence variables). Following Barendregt's Variable Convention [Bar84], bound variables are fresh and distinct from free variables.

Given a unification problem $t_1 =_\alpha^? t_2$, the core algorithm tries to find a substitution $\sigma$ such that $t_1\sigma =_\alpha t_1\sigma$, where $\sigma$ may instantiate only free variables (so called meta-variables) of $t_1$ and $t_2$. Individual meta-variables are mapped to individual expressions. Sequence meta-variables are mapped to finite sequences of expressions. In the process of unification, bound variables can be renamed into other bound variables.

As an example, consider the unification problem between two statements about sets: $X + 1 \in \{x \mid_x x > X\} =_\alpha^? Y \in \{y \mid_y y > a\}$, where $X$ and $Y$ are individual meta-variables. The algorithm computes the unifier $\sigma = \{Y \mapsto a + 1, X \mapsto a\}$. In the process of computing $\sigma$, the bound variable $x$ is renamed into the bound variable $y$.

Substitutions avoid variable capture. That implies that, for instance, if $\mathsf{Q}$ is some quantifier and $X$ is a meta-variable, the unification problem $\underset{x}{\mathsf{Q}}\, X =_\alpha^? \underset{y}{\mathsf{Q}}\, y$ is unsolvable. In this way, the unification algorithm implemented in Theorema differs from nominal unification [UPG04] that also tries to unify terms modulo $\alpha$-equivalence, but permits variable capture.

The reasoners that invoke the unification function should make sure that reasoning with quantifiers is sound. Therefore, they pass the unification algorithm information about dependencies between meta-variables and arbitrary but fixed constants (sometimes also called parameters) that have been introduced by some rules dealing with quantifiers. For instance, a prover can reduce the problem of proving the formula $\underset{x}{\forall}\, p(x, x) \Rightarrow \underset{y}{\exists}\, \underset{z}{\forall}\, p(y, z)$ to the unification problem $p(X, X) =_\alpha^? p(Y, a_z)$, where $X$ and $Y$ are meta-variables (introduced by the corresponding quantifier rules in place of $x$ and $y$, respectively) and $a_z$ is a new arbitrary but fixed constant (also introduced by the corresponding quantifier rule in place of $z$). In addition, the information that $a_z$ depends on $Y$ is passed to the unification function. If this dependence were not there, the unification problem would have the solution $\{X \mapsto a_z, Y \mapsto a_z\}$. But the dependence forbids $Y$ to be mapped to $a_z$ and, therefore, there is no unifier. Depending on the prover, it is also possible that proving $\underset{x}{\forall}\, p(x,x) \Rightarrow \underset{y}{\exists}\, \underset{z}{\forall}\, p(y,z)$ gives raise to the unification problem $\underset{x}{\forall}\, p(x,x) =^?_\alpha \underset{z}{\forall}\, p(Y,z)$. Also here, the algorithm can not compute the solution. An attempt to replace $Y$ by $z$ would lead to the variable capture. Hence, the proof fails. All these are well-known techniques in quantifier reasoning.

Since unification is not done modulo the $\beta$-rule, the equation $X(a) =^?_\alpha f(a,a)$ does not have a solution (in contrast to four unifiers when the $\beta$-rule is permitted). Note also that the Theorema unification function does not see $f(a)(a)$ and $f(a,a)$ equal. The problem $X(a) =^?_\alpha f(a)(a)$ can be solved by $\{X \mapsto f(a)\}$. For reasoners with extra axioms about the expression equality, the core unification algorithm can be extended with the corresponding rules.

Sequence variables, a specific feature of the Theorema syntax, make the language very flexible, but as the price, unification becomes difficult. Some problems, such as, e.g. $f(a,\overline{X}) =^?_\alpha f(\overline{X},a)$, where $\overline{X}$ is a sequence variable, can have infinitely many incomparable unifiers: $\{\overline{X} \mapsto ()\}$, $\{\overline{X} \mapsto a\}$, $\{\overline{X} \mapsto (a,a)\}, \ldots$. This obviously causes a challenge to reasoning, but there are ways to deal with it. On the one hand, finitary fragments of unification with sequence variables have been identified, see e.g. [GF92, RF97, Kut03, Kut07, KM12]. On the other hand, constraint- and expansion-based reasoning methods have been proposed, e.g. [Gin91, Kut02b, HV06, PB13]. The unification algorithm of Theorema utilizes both ideas: It tries to solve the unification problem (that involves sequence variables) completely with the help of the algorithm introduced in [Kut02a]. In this process, if an equation appears that does not belong to the known finitary fragments, the algorithm switches to the incomplete method of expansion, replacing sequence variables by sequences of fresh individual variables up to a certain predefined length. For instance, if this length is 3, for the above mentioned unification problem $f(a,\overline{X}) =^?_\alpha f(\overline{X},a)$ the algorithm will return four unifiers: $\{\overline{X} \mapsto ()\}$, $\{\overline{X} \mapsto a\}$, $\{\overline{X} \mapsto (a,a)\}$, and $\{\overline{X} \mapsto (a,a,a)\}$.

Besides the maximal length of expansion sequences, the unification function has another parameter: the maximal number of solutions to be computed. Sequence variables may lead to multiple solutions, and with this parameter one can easily control the desired number of them.

When talking about specific features of the language, we should note that, by default, $f$ and $f()$ are not identified. Therefore, the unification problem $f =^?_\alpha f(\overline{X})$ does not have a solution, while $f() =^?_\alpha f(\overline{X})$ has one, namely the substitution $\{\overline{X} \mapsto ()\}$. However, there is an option to extend the core algorithm to work with the equational theory specified by the equality $f \doteq f()$ for some $f$ (or for all symbols).

Yet another equational theory, to which the core algorithm has been extended, is generalized commutativity, or, in other terms, the theory of orderless symbols. A symbol is orderless if the order of its arguments does not matter. The unification function has an option to set the orderless property for certain function symbols, and then apply the respective rules for unification. For efficiency reasons, sequence variables under orderless symbols are treated by the expansion method (although there exists a complete unification procedure, based on the reduction to systems of linear Diophantine equations [Kut02a]).

## 5. ALGORITHM SYNTHESIS BASED ON LOGICAL REASONING

Here, we consider a very general version of the notion of *solving*: given a problem specification, find an algorithm that, when applied to any input that satisfies the input condition, computes a result that satisfies the output condition. In this context, a *problem specification* consists of an input condition and an output condition. As an example, the problem of "sorting" can be described by the input condition "the input is a list" and the output condition "the output is the sorted version of the input". *Solving the sorting problem* amounts to *finding an algorithm* that computes the sorted version of its input. In other words, we need to find a sorting algorithm.

This section summarizes a case study in *Theorema* on synthesis of sorting algorithms in parallel with the exploration of the theory of lists. For additional details, see [DJ15]. The use of *Theorema* facilitates this work because the system supports the implementation of complex inference rules as described in Section 3.2 and also because the formulae and the proofs are presented in natural style, making it easy to understand the development of the theory. The purpose of this case study is to demonstrate the use of mechanical proving in the automated development of non-trivial algorithms. Moreover we identify special inference rules and specific strategies for automated reasoning in the domain of lists. This work extends [BC04b] by using part of the developed theory (however modified in order to be first order) and by introducing a different approach for expressing algorithmic ideas (logical formulae instead of algorithm schemata).

The relevance of this case study for the QED-related activities consists in the development of a certain theory and of an arsenal of proving methods for the purpose of solving a certain practical problem. This mimics the process of mathematical progress on a simple example: we start from the concept of list and of sorting, and try to find the appropriate axioms and definitions which are necessary for the formalization of these concepts. Then we attempt to synthesize sorting algorithms by proving, where new properties and notions and also some specific proof methods turn out to be necessary.

### 5.1 Approach

The approach consists of extracting the algorithm from the (constructive) proof of the statement: "For any input satisfying the input condition, there exists an object satisfying the output condition between the input and the output". The input and the output condition of the function to be implemented constitute the specification, which is the input to the synthesis process. The output of the synthesis is a collection of conditional equalities which can be used to compute the desired function. The proof needs a corresponding theory of the domains which are relevant for the problem (that is, a collection of logical formulae).

In the simplest case, a non-recursive algorithm for the target function exists, the functions necessary for the implementation of the algorithm are already present in the theory. In this case the proof will exhibit a set of witness terms which expresses the algorithm in non-recursive way—possibly with conditions. In the sequel we will call such sets of conditioned witnesses simply *witness*. (In fact, Theorema offers a special notation for expressing such sets of conditioned terms as a single term.) For instance, in our case study, such an algorithm is the one for computing the maximum of two values.

If some of the functions necessary for the algorithm are not in the theory, then the proof fails and from the failed proof one can identify the specification of one or more of the missing auxiliary functions. This approach was pioneered in [BC04b], which also automated a certain heuristic for the production of such specifications. (The case study presented here offers a basis for further automation of this process.) The algorithm synthesis process continues with the finding of the necessary algorithms based on these new specifications, and if still some of them are not found, the original proof is re-iterated (with more success this time), in order to find more specifications of more functions. For instance, in our case study, the quick-sort algorithm requires an auxiliary function for splitting a list in two, such that each element in the first list is smaller than each element in the second list.

Recursive algorithms can be discovered by inductive proofs (see e.g. [BSW90]). We follow the approach that the algorithmic idea for the desired implementation is presented in the form of an induction principle, consisting of a composition function and the corresponding decomposition functions. (The base cases correspond to the objects which are too small to be decomposed.) The proof branches for the base cases yield simple ground witnesses, and the inductive step produces a witness which is dependent (in our natural style proving) on some Skolem constants. Some of these constants correspond to the recursive calls of the algorithm on the parts of the decomposed argument. In our case study we use two induction principles: *head-tail* (as in Lisp programming) and *split* (decompose the list in two parts, compose a list from two lists by concatenation). For the problem of sorting, the base case consists naturally of empty and unit lists in both induction principles.

Since we work in first order logic[7], it is sure that if a non-recursive algorithm exists, then it will be found. Completeness of first order proving insures that a set of witness terms is found, which corresponds to the provable disjunction of ground instances of the existential goal. When the set is non-unitary, then this expresses an "if-then-else" structure of the algorithm, where the respective conditions can also be extracted from the proof. We developed a special strategy for the extraction of such conditional algorithms: identification and use of *elementary goals*. An elementary goal is a formula which cannot be reduced anymore and it contains only functions which can be evaluated in constant time. In this case the proof is considered successful on the current branch and the current elementary goal becomes the condition for the current witness. The prover creates another branch of the proof (at the suitable previous point in the deduction), in which the negation

[7]Using induction in first order logic is possible because the induction principle is already instantiated with the property which we want to prove, and moreover the induction is often used as an inference rule.

of the respective elementary formula is assumed. In this way further conditional branches of the algorithm will be found. For instance, in our case, such a non-recursive conditional algorithm is the maximum of two numbers. The technique of identifying branching parts of the algorithms is used however also in the case of non-recursive algorithms, for identifying witnesses on the inductive branches of the proofs. For instance, in our case study, such an "if-then-else" witness is found on the inductive branch of the proof synthesizing the merging algorithm (merge two sorted lists into a sorted one), which is the auxiliary function needed for the merge-sort algorithm.

In addition to the induction principle, we show how one can specify algorithmic ideas by logical formulae (which become the goals on the inductive branches). In the following, $O$ denotes the output condition of the specification of the target function (an input predicate is not necessary in this case), while $C, C'$ denote some composition functions on lists. All variables are of type "list". The algorithmic idea "we decompose the input using the reverse of $C$, then we apply recursively the target function on the parts, and then we use $C'$ to compose the desired result from the partial results" can be expressed as:

$$\underset{X}{\forall}\ \underset{X_1,X_2}{\exists}\ C[X_1,X_2]=X \wedge \underset{Y_1,Y_2}{\forall}\ (O[X_1,Y_1]\wedge O[X_2,Y_2]) \Rightarrow \\ \underset{Y}{\exists}\ Y=C'[Y_1,Y_2]\wedge O[X,Y]$$

If the composition function $C$ for the input is known (for instance, if it is the one defined by the induction principle), then the synthesis of the algorithm consists in finding the appropriate composition function for the output from the proof of the following simplified formula:

$$\underset{X_1,X_2}{\forall}\ \underset{Y_1,Y_2}{\forall}\ (O[X_1,Y_1]\wedge O[X_2,Y_2]) \Rightarrow \underset{Y}{\exists}\ O[C[X_1,X_2],Y]$$

The witness $T[X_1,X_2,Y_1,Y_2]$ for the existential variable $Y$ will define the recursion of the desired algorithm by a pattern matching equality $F[C[X_1,X_2]] = T[X_1,X_2,F[X_1],F[X_2]]$. This can be transformed into a functional equality by using $F[X]$ on the left-hand side and the (induction corresponding) decomposition of $X$ into $X_1,X_2$ on the right-hand side.

If the composition function $C'$ for the output is known (for instance, if it is the one defined by the induction principle), then the synthesis of the algorithm consists in finding the appropriate decomposition function for the input from the proof of the following simplified formula:

$$\underset{X}{\forall}\ \underset{X_1,X_2}{\exists}\ \underset{Y_1,Y_2}{\forall}\ (O[X_1,Y_1]\wedge O[X_2,Y_2]) \Rightarrow O[X,C'[Y_1,Y_2]]$$

The witnesses for the existential variables $X_1,X_2$ will express the required decomposition algorithm.

## 5.2 Experiments

We followed the principles described above by using as implicit input condition $I[X]$: "$X$ is a list" and as output condition $O[X,Y]$: "$Y$ is the sorted version of

$X$". The later is composed of two atoms: "$X$ is a permutation of $Y$" and "$Y$ is sorted".

The first version of the list theory contained the definitions of these two atoms and the definitions of the necessary notions corresponding to them (like e.g. the ordering relation among the elements of the lists). After that, certain properties have been added (after being mechanically proven), like e.g. the fact that "is a permutation of" is an equivalence relation, and the properties of the ordering. During the first proof attempts it also became clear that certain additional predicates are useful, for instance the comparison between an element and a list (the element is smaller than each element in the list, or the other way around), and similarly between two lists. More properties have been added for these new predicates as well as for the interaction between them and "is sorted". The theory exploration continued with the addition of the necessary auxiliary functions which have been synthesized during the process.

The synthesis proceeded with repeated attempts to prove the main statement ("a sorted version of the input list exists") in an inductive way using several branches:

—two branches for the base cases (empty list and unit list);

—two branches for the *head-tail* induction principle:

  — known input decomposition (head-tail) and unknown output composition: this yields the insert-sort algorithm, the auxiliary composition function inserts an element into a sorted list;

  — unknown input decomposition and known output composition ("cons" of Lisp): this yields the max-sort algorithm, the auxiliary function finds the maximum of a list;

—three branches for the *split* induction principle:

  — known input decomposition (split) and unknown output composition: this yields the merge-sort algorithm, the auxiliary composition function merges two sorted lists into a sorted list;

  — unknown input decomposition and known output composition (concatenation): this yields the quick-sort algorithm, the auxiliary function splits a list such that each element in the first is smaller than each element in the second;

  — both decomposition and composition are unknown, such that only the second list has to be sorted: this yields a novel algorithm which we call "unbalanced merge sort": at decomposition the first list is constructed by taking the elements which are already in the appropriate order, while the other elements are put in the second list; for output composition the same merging function as in the first subbranch of this group is used.

In this way we finally generated 5 sorting algorithms, as well as the necessary auxiliary functions. All proofs for the synthesis and for the required properties are generated automatically[8] (more than 50 proofs). The synthesis proceeded iteratively, by conjecturing properties of the auxiliary functions from failed proofs, and by adding and proving the necessary properties. The proofs are performed in natural style, using predicate logic inference rules (instantiation, modus ponens,

[8]The human interaction for every proof consists in selecting the assumptions which are necessary for the proof and some options of the prover (e. g. which induction principle should be used).

back-chaining on goal à la Prolog, Skolem constants, meta-variables) as well as specific inference rules and strategies for the domain of lists of elements from an ordered domain. These specific techniques reduced the search space by a significant factor. For instance, it was very useful to introduce a normal form for expressions containing concatenation over elements and lists: a pair of multi-sets, one for elements and one for lists. This allows efficient reduction of goals containing such expressions in atoms like "is permutation of" and "is sorted".

An example of a useful strategy is the consideration of what we call "micro-atoms" (ground atoms without function symbols). For instance an atom consisting of "is sorted" applied to a composite term can be decomposed in a maximal (as many as possible) or in a minimal (as few as necessary) number of micro-atoms. We decompose the goal minimally and the assumptions maximally, and then a significant reduction of the goal can be obtained.

## 6. COMPLEXITY ANALYSIS OF GRÖBNER BASES COMPUTATION

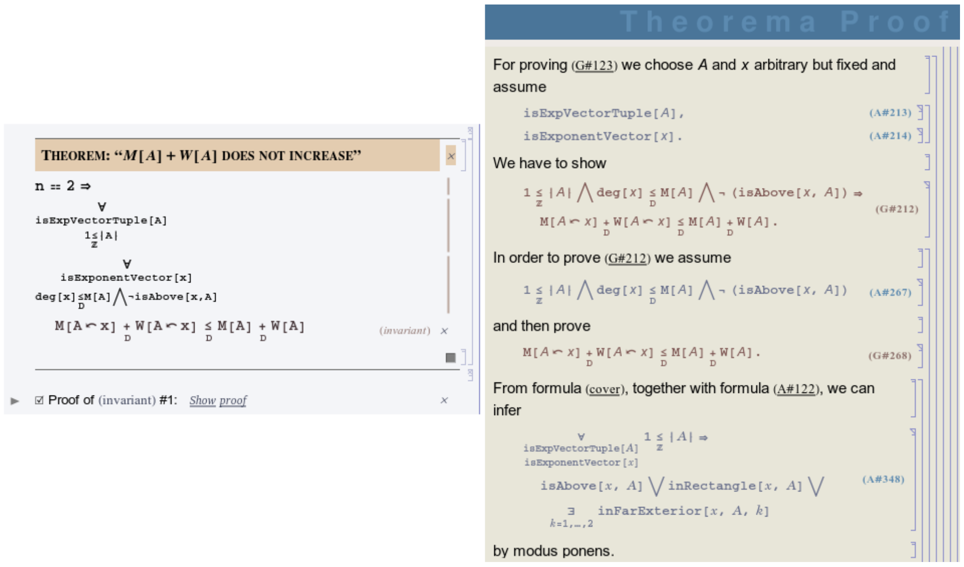
This section presents a case study in mathematical theory exploration which was recently carried out in Theorema 2.0. It is concerned with the complexity analysis of Buchberger's algorithm (with chain criterion) for computing Gröbner bases, restricted to *bivariate* polynomial rings. The algorithm itself, its improvement (chain criterion), as well as a pencil-and-paper elaboration of the complexity analysis presented here are all due to the first author [Buc65, Buc83]. The section is only meant to be an overview for illustrating how the formal development of a theory in Theorema typically proceeds, not as an in-depth treatment of the topic; more details can be found in [Mal14] instead.

Here, *complexity analysis* means verifying a certain upper bound on the degrees of the polynomials in the Gröbner basis computed by the aforementioned algorithm, depending on the degrees of the input polynomials. As soon as such a bound is known, the number of operations that have to be executed during the algorithm can easily be estimated (but this is *not* covered by the analysis presented here). One of the most important results needed for verifying the upper bound is the fact that a certain numeric quantity is an *invariant* of the algorithm, in the sense that it does not *increase* (but it may well decrease). This quantity only depends on the exponent vectors of the leading monomials of the polynomials in the so-far computed set $A$ in the course of the algorithm. The Theorema formula, which states that it does not increase when adding a new element $x$ (with certain properties) to $A$, is shown in Figure 5.[9]

### 6.1 Theory Exploration

The theory we explored is mostly concerned with *exponent vectors* and tuples thereof. An exponent vector can be thought of as a finite vector of natural numbers, where typical operations on exponent vectors are, for instance, component-wise addition of two vectors, divisibility of one vector w. r. t. another (defined to hold if and only if each component of the first vector is at least as big as the corresponding component of the other vector), and the least-common-multiple of two vectors.

[9]For technical reasons, we based our formalizations on *tuples* rather than *sets*. Hence, in Figure 5 $A$ is a tuple and $A \curvearrowleft x$ denotes the tuple $A$ with element $x$ appended.



THEOREM: "$M[A] + W[A]$ DOES NOT INCREASE"

$$n == 2 \Rightarrow \underset{\substack{\text{isExpVectorTuple}[A] \\ 1 \underset{\mathbb{Z}}{\leq} |A|}}{\forall} \; \underset{\substack{\text{isExponentVector}[x] \\ \deg[x] \underset{D}{\leq} M[A] \bigwedge \neg\text{isAbove}[x,A]}}{\forall} M[A \leftharpoonup x] \underset{D}{+} W[A \leftharpoonup x] \underset{D}{\leq} M[A] \underset{D}{+} W[A] \quad \text{(invariant)}$$

☑ Proof of (invariant) #1: Show proof

Theorema Proof

For proving (G#123) we choose $A$ and $x$ arbitrary but fixed and assume

$\text{isExpVectorTuple}[A]$, (A#213)

$\text{isExponentVector}[x]$. (A#214)

We have to show

$$1 \underset{\mathbb{Z}}{\leq} |A| \bigwedge \deg[x] \underset{D}{\leq} M[A] \bigwedge \neg(\text{isAbove}[x, A]) \Rightarrow M[A \leftharpoonup x] \underset{D}{+} W[A \leftharpoonup x] \underset{D}{\leq} M[A] \underset{D}{+} W[A]. \quad \text{(G\#212)}$$

In order to prove (G#212) we assume

$$1 \underset{\mathbb{Z}}{\leq} |A| \bigwedge \deg[x] \underset{D}{\leq} M[A] \bigwedge \neg(\text{isAbove}[x, A]) \quad \text{(A\#267)}$$

and then prove

$$M[A \leftharpoonup x] \underset{D}{+} W[A \leftharpoonup x] \underset{D}{\leq} M[A] \underset{D}{+} W[A]. \quad \text{(G\#268)}$$

From formula (cover), together with formula (A#122), we can infer

$$\underset{\substack{\text{isExpVectorTuple}[A] \\ \text{isExponentVector}[x]}}{\forall} 1 \underset{\mathbb{Z}}{\leq} |A| \Rightarrow \text{isAbove}[x, A] \bigvee \text{inRectangle}[x, A] \bigvee \underset{k=1,\ldots,2}{\exists} \text{inFarExterior}[x, A, k] \quad \text{(A\#348)}$$

by modus ponens.

Fig. 5. Theorema 2.0 formula stating that $M(A) + W(A)$ does not increase when adding a new element $x$ to $A$ (left), together with a fragment of its Theorema-generated proof (right).

Listed below are the individual steps we went through in the formal exploration of the theory:

(1) *Introduce Basic Notions*: At the very beginning, some of the frequently used basic notions the theory is built upon were defined, including the notions of tuples, linear order relations and commutative monoids. None of these notions is specific to the complexity analysis and may thus be used as a building block for other theories, too. Typically, such notions are collected in separate Theorema knowledge archives, see Section 2.2.2, so that they can readily be re-used in other explorations. An example of such a basic notion is that of a *total order relation*.

(2) *Prove Properties of Basic Notions*: Then, some (again very elementary) properties of the previously introduced notions were proved. For instance, from the defining properties of *reflexive* total order relations we formally verified that most properties (like transitivity) also hold for their strict, non-reflexive versions.

(3) *Create Special Prover*: This step is one of the most important ones and described in more detail in the next subsection. The main idea is to "lift" parts of the knowledge about the basic notions to the level of inference, such that later one does not have to fall back to the object-level formulae, but can directly use their semantic contents in a more convenient and efficient way.

(4) *Introduce Specific Notions*: Similar to the first step, new notions and concepts were introduced, but this time specific notions directly related to the complexity analysis, for instance the *chain criterion* predicate (which is a relation between two exponent vectors and a tuple of exponent vectors).

(5) *Prove Properties of Specific Notions*: Now, similar to the second step, some more or less elementary properties of the newly introduced specific notions were

proved, making use of the already acquired knowledge about the basic notions and the special prover that was created in the third step. For instance, we proved an alternative description of the chain criterion in case all exponent vectors only have two components.

(6) *Prove Main Theorems*: Finally, as the last step the main theorems the complexity analysis is concerned with were proved, making use of all the knowledge about basic and specific notions, as well as the special prover.

Most of the lemmas proved in steps (2) and (5) are so-called *rewrite-kind* formulae. A rewrite-kind formula is a (universally quantified, conditional) equality or equivalence, which can be used as a rewrite-rule for simplifying expressions. As it turned out, many theorems can be proved using only these rewrite-kind lemmas together with basic inference rules from predicate logic, resembling the way a human mathematician would carry out the proofs.

Nevertheless, in some situations rewrite-kind formulae were not sufficient. Therefore, we created a special prover in order to obtain short and natural proofs. This is described in the next subsection.

## 6.2 Creating New Provers

Creating special provers for individual theories has ever since been an integral part of the philosophy of Theorema: Theorema allows, if not encourages, the user exploring a certain theory to create a new prover that incorporates all the basic knowledge the theory is built upon, such that eventually proofs become short and simple and resemble proofs done by human mathematicians.

More precisely, the term "special prover" must be understood in a broad sense: A special prover is usually described by a collection of special inference rules that behave particularly well when working in certain theories. They may range from rules for newly-introduced quantifiers (note that Theorema does not yet support higher-order rewriting by object-level formulae at the moment!) over rules that simply apply pre-defined rewrite rules following a *control* that is different from the ones used by the default rewriting mechanisms, to inference rules that enhance the logic of Theorema (e.g. by adding types to expressions). Still, it must be clear that in most cases a special inference rule is merely an *abbreviation* that combines elementary inference techniques (simplification, etc.) into one compact step, see Section 3.2.

The special prover we created for the complexity analysis embodies precisely the aforementioned ideas. In particular, it consists of inference rules that in some sense "imitate" the presence of object-level formulae in the knowledge base. The advantage of such an approach can be illustrated best in a concrete example: consider the associative-commutative (AC) binary function $+$ and the linear order relation $\leq$ appearing in the theory. Not only is $+$ AC, but it is also monotonic w. r. t. $\leq$, in the sense that $z+x \leq z+y$ whenever $x \leq y$. This means that whenever we have to prove that a (nested) sum of elements $x_1, x_2, \ldots, x_n$ is less than or equal to some $y$, and we know that the (nested) sum of some of the $x_i$, say $x_{i_1}, \ldots, x_{i_m}$, is less than or equal to some $z$, then we can remove all the $x_{i_j}$ from the left-hand-side of the proof goal and add one single $z$ instead. Monotonicity and transitivity guarantee that if we manage to prove the new goal, also the original goal is proved. The problem now is as follows: although all the properties that are needed to carry out the inference, namely AC and monotonicity of $+$, can easily be stated as ordinary object-level formulae and can, hence, also be used in the knowledge base in proofs, often the resulting proofs become unnecessarily long and complicated. For instance, if $x_3 + x_1 \leq z$ is known and $(x_1 + x_2) + x_3 \leq y$ has to be shown, it is apparent that the definitions of associativity, commutativity and monotonicity have to be instantiated several times with different terms such that eventually the original goal can be replaced by $x_2 + z \leq y$. This is something one usually wishes to avoid, because on the one hand it makes proving inefficient, and on the other hand it distracts the user from the "more interesting" parts of the proof (typically, inferences like the one described here are standard transformations of proof situations a human mathematician (at this level of exploration) is not really interested in[10]). One convenient way to overcome this problem is to create a special inference rule which implicitly uses the AC- and monotonicity properties of $+$ on the meta-level for simplifying the proof goal to $x_2 + z \leq y$ *in one single step*, which is precisely the approach we pursued in our prover. For the sake of clarity it must be mentioned that it cannot only handle inequalities and monotonic functions, but also other concepts such as tuples and the minimum/maximum functions. A more detailed description would go beyond the frame of this paper but can be found in [Mal14] instead.

As already explained in Sections 2 and 3, Theorema provers have to be coded directly in the Mathematica programming language. Theorema neither provides any user-interface allowing them to be implemented in the Theorema language, nor is it possible to let Theorema carry out the lifting process described above (i.e. turn object-level formulae into inference rules) *automatically.* All this has to be done manually. Since we have not yet included quoting and reflection in Theorema 2.0, see Section 2 and [GB07], the Theorema provers cannot be verified within Theorema itself. Their validity must be justified by hand-written correctness proofs.

## 6.3 Remarks on the Formalization

The case study presented in this section demonstrates how a non-trivial mathematical theory can be formalized and verified in a natural but nonetheless rigorous way in Theorema 2.0. Furthermore, even though the theory already existed for more than 30 years and hence is by no means "new", it nevertheless profited from the formal treatment in the sense that some simplifications and generalizations (compared to the original pencil-and-paper elaboration) could be achieved. For more information on this, the interested reader is referred to [Mal14].

In order to get an impression of the size of the theory, we list some figures. The total number of formulae in the formalization is 292, with 62 being definitions and 230 being lemmas and theorems, hence, 230 proofs had to be generated. The new special prover consists of 57 inference rules whose implementation makes up 2650 lines of Mathematica code. The time needed to carry out the theory exploration

[10]It is very important to realize that what is important and interesting in a proof is not something that can be fixed once and for all—in particular it must not be fixed by the reasoning software. It always depends on the concrete phase of theory exploration, in which a user currently is. While AC-rewriting is certainly interesting for a user in a phase where AC is newly introduced, it is usually not interesting anymore in a phase where this concept is completely understood.

(including the design and implementation of the special prover) was, at a rough estimate, 300 hours.

## 7. CONCLUSION

This work reflects the current status of the Theorema project and its concrete implementation in form of the Theorema 2.0 system. We describe the new interface, the general system architecture and some promising first case studies that show the feasibility of the system's recent re-implementation. As future work, we will have to carry over special reasoning techniques that were already available in Theorema 1 (such as set theory proving or induction provers for various domains). Furthermore, we want to embark on the aspect of solving in the Theorema system. The approach described in Section 5 is a first step into this direction, but it is not yet entirely clear, how this will integrate into the Theorema system from a user's point of view. The issues to be settled include the design of an interface for specifying problems and for defining algorithm schemes together with appropriate induction principles. Last but not least, we also want to provide a rich library of knowledge archives to be used together with the basic system.

## References

[ARSCT11] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The Matita Interactive Theorem Prover. In Nikolaj Bjorner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23*, volume 6803 of *Lecture Notes in Computer Science*, pages 64–69. Springer Berlin Heidelberg, 2011.

[Bar84] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, Amsterdam, second, revised edition, 1984.

[BC04a] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer Verlag, 2004.

[BC04b] Bruno Buchberger and Adrian Craciun. Algorithm Synthesis by Lazy Thinking: Examples and Implementation in Theorema. In Fairouz Kamareddine, editor, *Electronic Notes in Theoretical Computer Science*, volume 93, pages 24–59, February 18 2004. Proc. of the Mathematical Knowledge Management Workshop, Edinburgh, Nov. 25, 2003.

[BCJ$^+$06] Bruno Buchberger, Adrian Craciun, Tudor Jebelean, Laura Kovacs, Temur Kutsia, Koji Nakagawa, Florina Piroi, Nikolaj Popov, Judit Robu, Markus Rosenkranz, and Wolfgang Windsteiger. Theorema: Towards Computer-Aided Mathematical Theory Exploration. *Journal of Applied Logic*, 4(4):470–504, 2006.

[BDJ$^+$00] Bruno Buchberger, Claudio Dupre, Tudor Jebelean, Franz Kriftner, Koji Nakagawa, Daniela Vasaru, and Wolfgang Windsteiger. The Theorema Project: A Progress Report. In Manfred Kerber and Michael Kohlhase, editors, *Symbolic Computation and Automated Reasoning (Proceedings of CALCULEMUS 2000, Symposium on the Integration of Symbolic Computation and Mechanized Reasoning)*, pages

98–113. St. Andrews, Scotland, Copyright: A.K. Peters, Natick, Massachusetts, August 6-7 2000.

[BJK$^+$97] Bruno Buchberger, Tudor Jebelean, Franz Kriftner, Mircea Marin, Elena Tomuta, and Daniel Vasaru. A Survey of the Theorema project. In W. Kuechlin, editor, *Proceedings of ISSAC'97 (International Symposium on Symbolic and Algebraic Computation, 1997)*, pages 384–391. ACM Press, 1997.

[BL79] B. Buchberger and F. Lichtenberger. *Mathematics for Computer Scientists I (German)*. Springer Verlag, Heidelberg, 1979.

[BMSS11] Ulrich Berger, Kenji Miyamoto, Helmut Schwichtenberg, and Monika Seisenberger. Minlog: A Tool for Program Extraction Supporting Algebras and Coalgebras. In *Proceedings of the 4th International Conference on Algebra and Coalgebra in Computer Science*, CALCO'11, pages 393–399, Berlin, Heidelberg, 2011. Springer-Verlag.

[BMTV97] Bruno Buchberger, Mircea Marin, Elena Tomuta, and Daniela Vasaru. Special Provers within the Theorema Project, April 28-30 1997. Contributed talk at CALCULEMUS'97 (International Workshop on Systems for Integrated Computation and Deduction), Trento, Italy.

[BSW90] Alan Bundy, Alan Smaill, and Geraint Wiggins. The Synthesis of Logic Programs from Inductive Proofs. In J. W. Lloyd, editor, *Computational Logic: Symposium Proc.*, pages 135–149. Springer, 1990.

[Buc65] Bruno Buchberger. *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal (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. English translation in J. Symb. Comp., Special Issue on Logic, Mathematics, and Computer Science: Interactions. Vol. 41, Number 3-4, Pages 475–511, 2006.

[Buc83] Bruno Buchberger. A Note on the Complexity of Constructing Gröbner-Bases. In J.A. van Hulzen, editor, *Computer Algebra (Proceedings of EUROCAL 83, European Computer Algebra Conference, London, March 28-30, 1983)*, volume 162 of *Lecture Notes in Computer Science*, pages 137–145. Copyright: Springer-Verlag Berlin–Heidelberg–New York–Tokyo, 1983.

[Buc96] Bruno Buchberger. Mathematica as a Rewrite Language. In Tetsuo Ida, Atsushi Ohori, and Masato Takeichi, editors, *Functional and Logic Programming (Proceedings of the 2nd Fuji International Workshop on Functional and Logic Programming, November 1-4, 1996, Shonan Village Center)*, pages 1–13. Copyright: World Scientific, Singapore - New Jersey - London - Hong Kong, 1996.

[Buc97] Bruno Buchberger. Mathematica: Doing Mathematics by Computer? In Alfonso Miola and Marco Temperini, editors, *Advances in the Design of Symbolic Computation Systems*, pages 2–20. Springer Vienna, 1997. RISC Book Series on Symbolic Computation.

[Buc04] Bruno Buchberger. Towards the Automated Synthesis of a Gröbner Bases Algorithm. *RACSAM—Revista de la Real Academia de Ciencias (Review of the Spanish Royal Academy of Science), Serie A: Mathematicas*, 98(1):65–75, 2004.

[DJ15] Isabela Dramnesc and Tudor Jebelean. Synthesis of list algorithms by mechanical proving. *J. Symb. Comput.*, 69:61–92, 2015.

[EJ10] Madalina Erascu and Tudor Jebelean. A Purely Logical Approach to the Termination of Imperative Loops. In T. Ida, V. Negru, T. Jebelean, D. Petcu, S. M. Watt, and D. Zaharie, editors, *Proceedings of SYNASC 2010: The 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 142–149. IEEE Computer Society, 2010.

[GAA$^+$13] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A Machine-Checked Proof of the Odd Order Theorem. In Sandrine Blazy, Christine Paulin, and David Pichardie, editors, *ITP 2013, 4th Conference on Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 163–179, Rennes, France, July 2013. Springer.

[GB07] Martin Giese and Bruno Buchberger. Towards Practical Reflection for Formal Mathematics. In *Proceedings of Austria-Japan Workshop on Symbolic Computation and Software Verification*, pages 30–34, Research Institute for Symbolic Computation (RISC), JKU Linz, Austria, 2007. Appeared as RISC Report 07-09. Extended abstract.

[GF92] Michael R. Genesereth and Richard E. Fikes. Knowledge Interchange Format. Version 3.0. Reference Manual. Technical Report KSL-92-86, Comp. Sci. Department, Stanford University, June 1992.

[Gin91] Matthew L. Ginsberg. The MVL Theorem Proving System. *SIGART Bulletin*, 2(3):57–60, 1991.

[Hal12] Thomas C. Hales. *Dense Sphere Packings: A Blueprint for Formal Proofs*. Number 400 in London Mathematical Society Lecture Note Series. Cambridge University Press, 2012.

[Har96] John Harrison. HOL Light: A Tutorial Introduction. In Mandayam Srivas and Albert Camilleri, editors, *Proceedings of the 1st International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer-Verlag, 1996.

[HV06] Ian Horrocks and Andrei Voronkov. Reasoning Support for Expressive Ontology Languages Using a Theorem Prover. In Jürgen Dix and Stephen J. Hegner, editors, *Foundations of Information and Knowledge Systems, 4th International Symposium, FoIKS 2006, Budapest, Hungary, February 14-17, 2006, Proceedings*, volume 3861 of *Lecture Notes in Computer Science*, pages 201–218. Springer, 2006.

[JBK+09] Tudor Jebelean, Bruno Buchberger, Temur Kutsia, Nikolaj Popov, Wolfgang Schreiner, and Wolfgang Windsteiger. Automated Reasoning. In B. Buchberger, M. Affenzeller, A. Ferscha, M. Haller, T. Jebelean, E.P. Klement, P. Paule, G. Pomberger, W. Schreiner, R. Stubenrauch, R. Wagner, G. Weiss, and W. Windsteiger, editors, *Hagenberg Research*, pages 63–101. Springer, 2009.

[KJ01] Boris Konev and Tudor Jebelean. Using Meta-variables for Natural Deduction in Theorema. In M. Kerber and M. Kohlhase, editors, *Calculemus 2000, St.Andrews, Scotland*. A.K. Peters, Natick, Massachusetts, 2001.

[KJ06] Laura Kovacs and Tudor Jebelean. Finding Polynomial Invariants for Imperative Loops in the Theorema System. In S. Autexier and H. Mantel, editors, *Proceedings of Verify'06 Workshop, IJCAR'06, The 2006 Federated Logic Conference*, pages 52–67, 2006.

[KM12] Temur Kutsia and Mircea Marin. Solving, Reasoning, and Programming in Common Logic. In Voronkov [Vor12], pages 119–126.

[KMM00] Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[Koh12] Michael Kohlhase. The Flexiformalist Manifesto. In Voronkov [Vor12], pages 30–35.

[Kut02a] Temur Kutsia. *Solving and Proving in Equational Theories with Sequence Variables and Flexible Arity Symbols.* PhD thesis, Johannes Kepler University, Linz, Austria, 2002.

[Kut02b] Temur Kutsia. Theorem Proving with Sequence Variables and Flexible Arity Symbols. In Matthias Baaz and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 9th International Conference, LPAR 2002, Tbilisi, Georgia, October 14-18, 2002, Proceedings*, volume 2514 of *Lecture Notes in Computer Science*, pages 278–291. Springer, 2002.

[Kut03] Temur Kutsia. Equational Prover of Theorema. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9-11, 2003, Proceedings*, volume 2706 of *Lecture Notes in Computer Science*, pages 367–379. Springer, 2003.

[Kut07] Temur Kutsia. Solving Equations with Sequence Variables and Sequence Functions. *J. Symb. Comput.*, 42(3):352–388, 2007.

[Mal14] Alexander Maletzky. Complexity Analysis of the Bivariate Buchberger Algorithm in Theorema. Technical Report 2014-10, Doctoral College "Computational Mathematics", October 2014.

[MR05] Roman Matuszewski and Piotr Rudnicki. Mizar: The First 30 Years. *Mechanized Mathematics and its Applications*, 4(1):3–24, 2005.

[MSW07] Günther Mayrhofer, Susanne Saminger, and Wolfgang Windsteiger. CreaComp: Experimental Formal Mathematics for the Classroom. In

Shangzhi Li, Dongming Wang, and Jing-Zhong Zhang, editors, *Symbolic Computation and Education*, pages 94–114, Singapore, New Jersey, 2007. World Scientific Publishing Co.

[PB13] Adam Pease and Christoph Benzmüller. Sigma: An Integrated Development Environment for Formal Ontology. *AI Commun.*, 26(1):79–97, 2013.

[PJ11] Nikolaj Popov and Tudor Jebelean. Sound and Complete Verification Condition Generator for Functional Recursive Programs. In U. Langer and P. Paule, editors, *Numerical and Symbolic Scientific Computing: Progress and Prospects*, pages 219–256. Springer, Wien, 2011.

[RF97] Julian Richardson and Norbert E. Fuchs. Development of Correct Transformation Schemata for Prolog Programs. In Norbert E. Fuchs, editor, *Logic Programming Synthesis and Transformation, 7th International Workshop, LOPSTR'97, Leuven, Belgium, July 10-12, 1997, Proceedings*, volume 1463 of *Lecture Notes in Computer Science*, pages 263–281. Springer, 1997.

[RK13] F. Rabe and M. Kohlhase. A Scalable Module System. *Information and Computation*, 230(1):1–54, 2013.

[RRTB11] Markus Rosenkranz, Georg Regensburger, Loredana Tec, and Bruno Buchberger. Symbolic Analysis for Boundary Problems: From Rewriting to Parametrized Gröbner Bases. In Ulrich Langer and Peter Paule, editors, *Numerical and Symbolic Scientific Computing: Progress and Prospects*, pages 273–331. Springer, Wien, 2011. Also available as RICAM Report 2010-05, September 2010.

[TB85] Andrzej Trybulec and Howard Blair. Computer Assisted Reasoning with Mizar. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 26–28, 1985.

[UPG04] Christian Urban, Andrew M. Pitts, and Murdoch Gabbay. Nominal unification. *Theor. Comput. Sci.*, 323(1-3):473–497, 2004.

[VJB09] Robert Vajda, Tudor Jebelean, and Bruno Buchberger. Combining Logical and Algebraic Techniques for Natural Style Proving in Elementary Analysis. *Mathematics and Computers in Simulation*, 79(8):2310–2316, April 2009. Special Issue on Nonstandard Applications of Computer Algebra.

[Vor12] Andrei Voronkov, editor. *14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2012, Timisoara, Romania, September 26-29, 2012.* IEEE Computer Society, 2012.

[Wie06] Freek Wiedijk. *The Seventeen Provers of the World: Foreword by Dana S. Scott (Lecture Notes in Computer Science / Lecture Notes in Artificial Intelligence).* Springer-Verlag New York, Inc., USA, 2006.

[Win01] Wolfgang Windsteiger. *A Set Theory Prover in Theorema: Implementation and Practical Applications.* PhD thesis, RISC Institute, 2001.

[Win12] Wolfgang Windsteiger. Theorema 2.0: A Graphical User Interface for a Mathematical Assistant System. In C. Kaliszyk and C. Lueth, editors, *Proceedings 10th International Workshop On User Interfaces for Theorem Provers, Bremen, Germany, July 11th 2012*, volume 118 of *Electronic Proceedings in Theoretical Computer Science*, pages 72–82. Open Publishing Association, 2012. doi 10.4204/EPTCS.118.5.

[WPN08] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle Framework. In Ait Mohamed, Munoz, and Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2008)*, number 5170 in LNCS, pages 33–38. Springer, 2008.

[WSB14] Max Wisnieski, Alexander Steen, and Christoph Benzmüller. The Leo-III Project. In Alexander Bolotov and Manfred Kerber, editors, *Joint Automated Reasoning Workshop and Deduktionstreffen*, page 38, 2014.