TH∃OREM∀: Extending Mathematica by Automated Proving

Bruno Buchberger
Research Institute for Symbolic Computation
University of Linz, Austria

Bruno.Buchberger@RISC.Uni-Linz.ac.at

Invited talk at the conference "The Programming System Mathematica in Science, Technology and Teaching"
Zagreb, September 27-28, 2001.

Acknowledgement: The research described in this paper is supported by the Austrian National Science Foundation (Fonds zur Förderung der Wissenschaftlichen Forschung, FWF) under contract number SFB 1302.

■ Introduction

Theorema is a system aiming at computer-supporting proving, solving, and simplifying as the three basic activities of formal mathematics. The system is programmed in the Mathematica programming language and, hence, is available on all platforms on wich Mathematica is installed. Theorema also heavily uses advanced features of the Mathematica front-end and it is also possible to access the complete computational functionality (the algorithm library) of Mathematica from the Theorema surface. Thus, *Theorema* can be seen as an extension of the typical current mathematical software systems towards making them a tool also for mathematical reasoning and mathematical knowledge management.

The main innovative feature of Theorema is its library of general and special automated theorem provers and its logical coherence: The Theorema language is a version of higher order predicate logic in attractive syntax, which comes close to the usual syntax of informal mathematics commonly used in mathematical textbooks and publications. The Theorema language allows to formulate non-algorithmic and algorithmic mathematics in one uniform frame. Part of the Theorema language is a universal programming language. Thus, formulae in Theorema can be processed either by provers (for deciding about the truth of formulae), or by solvers (for finding substitutions that make the formulae true) or by simplifyers (for finding equivalent simplified versions of the formulae). Collections of statements in the Theorema system can be organized in hierarchical knowledge bases that allow to express structure in mathematical knowledge and to access well-defined parts of the mathematical knowledge in a controlled way. The *Theorema* library of general and special provers, solvers, and simplifiers is permanently growing.

In this talk, we present Theorema by a sequence of live demo examples and we will then discuss the possible implications of systems like Theorema for the future of mathematics (research, teaching, applications).

The *Theorema* system is a joint effort of the *Theorema* Group at RISC under the direction of Bruno Buchberger with essential contributions by Tudor Jebelean, Wolfgang Windsteiger, Temur Kutsia, and Koji Nakagawa and the former and current PhD students Elena Tomuta, Franz Kriftner, Daniela Vasaru, Claudio Dupret, Florina Piroi, Markus Rosenkranz, and Christian Vogt.

■ Example: Manipulating Mathematical Text

We first give a couple of mathematical formulae in the syntax of *Theorema,* which essentially is a higher order predicate logic. The notation of *Theorema* is two-dimensional and should come as close as possible to the usual notation in mathematical text books. Before one can use *Theorema*, one must install the system and then load the system by entering

```
Needs["Theorema`"]
```

into a Mathematica cell.

The actual *Theorema* formulae are labeled by key words like 'Definition', 'Proposition', etc. and short text in quotation marks. These labels can be used later for referencing formulae. The 'any' construct declares the free variables in the formulae. All identifiers that are neither free nor quantified variables are constants. *Theorema* mathematical text like definitions, propositions etc. may be entered into

Definition["limit:", any[f, a],

$$\text{limit}[f, a] \Longleftrightarrow \underset{\substack{\epsilon \\ \epsilon>0}}{\forall}\ \underset{N}{\exists}\ \underset{\substack{n \\ n\geq N}}{\forall}\ |f[n] - a| < \epsilon\Big]$$

Proposition["limit of sum", any[f, a, g, b],
$(\text{limit}[f, a] \wedge \text{limit}[g, b]) \quad \Rightarrow \quad \text{limit}[f + g, a + b]]$

Definition["+:", any[f, g, x],
$(f + g)[x] = f[x] + g[x]]$

Lemma["|+|", any[x, y, a, b, δ, ϵ],
$(|(x + y) - (a + b)| < (\delta + \epsilon)) \quad \Longleftarrow \quad (|x - a| < \delta \wedge |y - b| < \epsilon)]$

Lemma["max", any[m, M1, M2],
$m \geq \max[M1, M2] \quad \Rightarrow \quad (m \geq M1 \wedge m \geq M2)]$

Now we illustrate how formulae can be composed into knowledge bases by the 'Theory' construct:

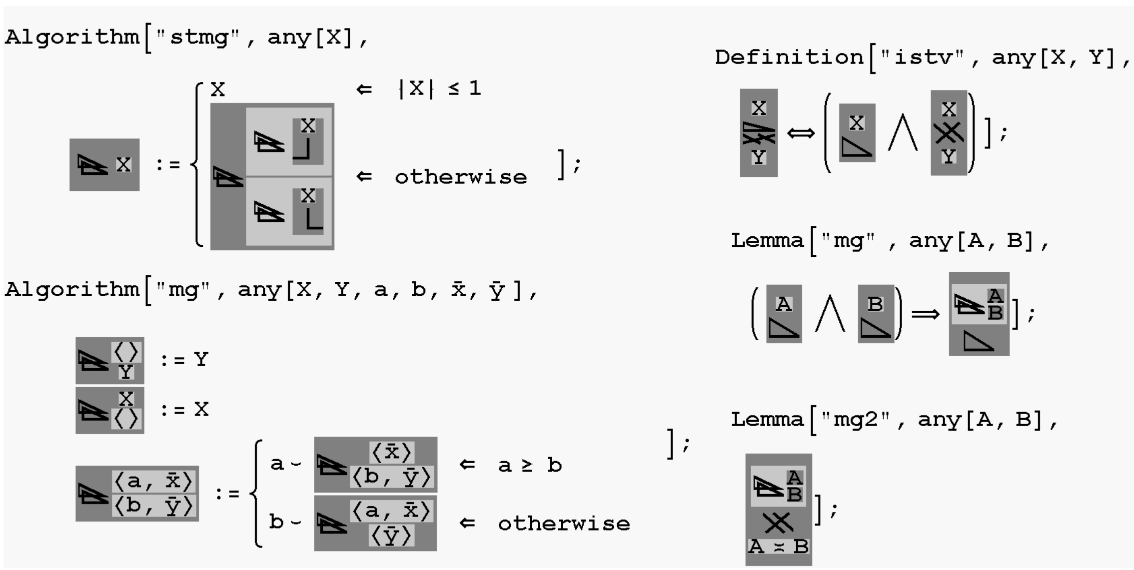
Theory["limit",
Definition["limit:"]
Definition["+:"]
Lemma["|+|"]
Lemma["max"]]

This knowledge base may later be referred to by Theory["limit"]. The 'Theory' construct is recursive. Thus, hierarchical knowledge bases can be composed.

Note that *Theorema* allows to express all formulae expressible in TeX and similar systems but is wysiwyg (like Mathematica) and, in contrast to ordinary mathematical text processing systems, produces formulae whose syntactical structure is entirely accessible so that formal mathematical manipulation (proving, simplifying, solving) can be done with such formulae.

■ Logicographic Symbols

In addition, and in contrast to all existing mathematical systems, *Theorema* provides the facility to design and introduce arbitrary new symbols for function and predicate constants, which may convey the intuitive meaning of these symbols by arbitrary complex and user-defined graphics. We call these symbols "logicographic symbols". An example of some *Theorema* formal text using logicographic symbols is given below. This text forms part of a theory of tuples on which the correctness proof of the merge-sort algorithm can be based.



The above presentation of Algorithm["stmg"] (sorting by merging) states that the sorting a tuple X produces X if the length of X is less or equal 1 and, otherwise, proceeds by merging the sorted left part of the sorted right part of X. Similarly, Algorith["mg"] (merging) is defined recursively using various logicographic symbols whose meaning should be self-explanatory or, at least, easy to remember. Definition["istv"] (is sorted version) states that tuple X is a sorted version of tuple Y iff X is sorted and X is a permuted version of Y. Lemma["mg"] (a property of merge) states that, if tuple A is sorted and tuple B is sorted, then the merge of A and B is also sorted. Lemma["mg2"] (another property of merge) states that the merge of A and B is a permuted version of the concatenation of A and B. Most of the symbols appearing in the above formulae are *not* in the fixed arsenal of symbols of Mathematica but can be designed and then declared and used by the *Theorema* user applying a couple of special tools provided by *Theorema*.

■ Example: The PCS Prover

The PCS proof method, proposed and developed by the author in 2000, is a heuristic proof method for predicate logic whose main goal is to generate "natural" proofs.

Roughly, the PCS prover proceeds by iteratively going through the following three phases:

- P-phase ("Proving" phase)

- C-phase ("Computing" phase, simplifying phase)

- S-phase ("Solving" phase).

The PCS method is particularly useful for proving theorems about notions defined by alternating quantifiers and reduces proving of such formulae to constraints solving. A typical example of a notion defined by alternating quantifiers is the notion of limit in analysis, see the definition above.Here is the call that calls the PCS prover for proving the above Proposition["limit of sum"] using the knowledge in the Theory["limit"]:

Prove[Proposition["limit of sum"], using → Theory["limit"], by → PCS]

After a couple of seconds, the following proof text (including the intermediate English explanatory text) will be produced completely automatically.

Prove:

(Proposition (limit of sum)) $\underset{f,a,g,b}{\forall}(\text{limit}[f, a] \wedge \text{limit}[g, b] \Rightarrow \text{limit}[f + g, a + b])$,

under the assumptions:

(Definition (limit:)) $\underset{f,a}{\forall}\left(\text{limit}[f, a] \Leftrightarrow \underset{\substack{\epsilon \\ \epsilon>0}}{\forall}\ \underset{N}{\exists}\ \underset{\substack{n \\ n\geq N}}{\forall}(|f[n] - a| < \epsilon)\right)$,

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

(Lemma (|+|)) $\underset{x,y,a,b,\delta,\epsilon}{\forall}(|(x + y) - (a + b)| < \delta + \epsilon \Leftarrow (|x - a| < \delta \wedge |y - b| < \epsilon))$,

(Lemma (max)) $\underset{m,M1,M2}{\forall}(m \geq \max[M1, M2] \Rightarrow m \geq M1 \wedge m \geq M2)$.

We assume

(1) $\text{limit}[f_0, a_0] \wedge \text{limit}[g_0, b_0]$,

and show

(2) $\text{limit}[f_0 + g_0, a_0 + b_0]$.

Formula (1.1), by (Definition (limit:)), implies:

(3) $\underset{\substack{\epsilon \\ \epsilon>0}}{\forall}\ \underset{N}{\exists}\ \underset{\substack{n \\ n\geq N}}{\forall}(|f_0[n] - a_0| < \epsilon)$.

By (3), we can take an appropriate Skolem function such that

(4) $\underset{\substack{\epsilon \\ \epsilon>0}}{\forall}\ \underset{\substack{n \\ n\geq N_0[\epsilon]}}{\forall}(|f_0[n] - a_0| < \epsilon)$,

Formula (1.2), by (Definition (limit:)), implies:

(5) $\underset{\substack{\epsilon \\ \epsilon>0}}{\forall} \underset{N}{\exists} \underset{\substack{n \\ n\geq N}}{\forall} (|g_0[n] - b_0| < \epsilon)$.

By (5), we can take an appropriate Skolem function such that

(6) $\underset{\substack{\epsilon \\ \epsilon>0}}{\forall} \underset{\substack{n \\ n\geq N_1[\epsilon]}}{\forall} (|g_0[n] - b_0| < \epsilon)$,

Formula (2), using (Definition (limit:)), is implied by:

(7) $\underset{\substack{\epsilon \\ \epsilon>0}}{\forall} \underset{N}{\exists} \underset{\substack{n \\ n\geq N}}{\forall} (|(f_0 + g_0)[n] - (a_0 + b_0)| < \epsilon)$.

We assume

(8) $\epsilon_0 > 0$,

and show

(9) $\underset{N}{\exists} \underset{\substack{n \\ n\geq N}}{\forall} (|(f_0 + g_0)[n] - (a_0 + b_0)| < \epsilon_0)$.

We have to find N_2^* such that

(10) $\underset{n}{\forall} (n \geq N_2^* \Rightarrow |(f_0 + g_0)[n] - (a_0 + b_0)| < \epsilon_0)$.

Formula (10), using (Definition (+:)), is implied by:

(11) $\underset{n}{\forall} (n \geq N_2^* \Rightarrow |(f_0[n] + g_0[n]) - (a_0 + b_0)| < \epsilon_0)$.

Formula (11), using (Lemma (|+|)), is implied by:

(12) $\underset{\substack{\delta,\epsilon \\ \delta+\epsilon=\epsilon_0}}{\exists} \underset{n}{\forall} (n \geq N_2^* \Rightarrow |f_0[n] - a_0| < \delta \wedge |g_0[n] - b_0| < \epsilon)$.

We have to find δ_0^*, ϵ_1^*, and N_2^* such that

(13) $(\delta_0^* + \epsilon_1^* = \epsilon_0) \bigwedge \underset{n}{\forall} (n \geq N_2^* \Rightarrow |f_0[n] - a_0| < \delta_0^* \wedge |g_0[n] - b_0| < \epsilon_1^*)$.

Formula (13), using (6), is implied by:

$$(\delta_0^* + \epsilon_1^* = \epsilon_0) \bigwedge \underset{n}{\forall} (n \geq N_2^* \Rightarrow \epsilon_1^* > 0 \wedge n \geq N_1[\epsilon_1^*] \bigwedge |f_0[n] - a_0| < \delta_0^*),$$

which, using (4), is implied by:

$$(\delta_0^* + \epsilon_1^* = \epsilon_0) \bigwedge \underset{n}{\forall} (n \geq N_2^* \Rightarrow \delta_0^* > 0 \wedge \epsilon_1^* > 0 \wedge n \geq N_0[\delta_0^*] \bigwedge n \geq N_1[\epsilon_1^*]),$$

which, using (Lemma (max)), is implied by:

$$(14) \quad (\delta_0^* + \epsilon_1^* = \epsilon_0) \bigwedge \underset{n}{\forall} (n \geq N_2^* \Rightarrow \delta_0^* > 0 \wedge \epsilon_1^* > 0 \wedge n \geq \max[N_0[\delta_0^*], N_1[\epsilon_1^*]]).$$

Formula (14) is implied by

$$(15) \quad (\delta_0^* + \epsilon_1^* = \epsilon_0) \bigwedge \delta_0^* > 0 \bigwedge \epsilon_1^* > 0 \bigwedge \underset{n}{\forall} (n \geq N_2^* \Rightarrow n \geq \max[N_0[\delta_0^*], N_1[\epsilon_1^*]]).$$

Partially solving it, formula (15) is implied by

$$(16) \quad (\delta_0^* + \epsilon_1^* = \epsilon_0) \wedge \delta_0^* > 0 \wedge \epsilon_1^* > 0 \wedge (N_2^* = \max[N_0[\delta_0^*], N_1[\epsilon_1^*]]).$$

Now,

$$(\delta_0^* + \epsilon_1^* = \epsilon_0) \wedge \delta_0^* > 0 \wedge \epsilon_1^* > 0$$

can be solved for δ_0^* and ϵ_1^* by a call to Collins cad–method yielding a sample solution

$$\delta_0^* \leftarrow \frac{\epsilon_0}{2},$$

$$\epsilon_1^* \leftarrow \frac{\epsilon_0}{2}.$$

Furthermore, we can immediately solve

$$N_2^* = \max[N_0[\delta_0^*], N_1[\epsilon_1^*]]$$

for N_2^* by taking

$$N_2^* \leftarrow \max[N_0[\tfrac{\epsilon_0}{2}], N_1[\tfrac{\epsilon_0}{2}]].$$

Hence formula (16) is solved, and we are done.

□

We hope that, from the sample proof, it is clear what we mean by a "natural" proof: It is structured very much the way a proof produced by a human mathematician would be structured and it should be "easy to read and understand". Note that, in addition, the proof generated by our PCS method also produces interesting information that is normally not produced in the proofs given in the usual mathematical textbooks: The explicit term for N_2^* explains in detail how an index bound for the sequence $f+g$ can be obtained from an index bound for the input sequences f and g. In a live session of *Theorema*, additional tools are available that support easy understanding of the proof. For example, the label hyperlinks can be used in order to open a small window in which the formula referenced will be displayed. Also, on the right-hand margin of the proof notebook, nested brackets are shown that make it possible to contract subproofs to just one line so that it is easy to preserve overview on the overall structure of the proof while studying certain details.

In fact, as simple as they may seem for well-trained mathematicians, proofs in elementary analysis are still a challenge for automated proving methods. Hence, the possibility to generate such proofs automatically and, in addition, come up with natural and readable proofs in this area, is quite a strong indication of the potential of the PCS method.

■ Example: Set Theory Prover

The *Theorema* set theory prover is also based on the PCS principle. In addition to the predicate logic natural deduction inference rules it incorporates special inference rules for the basic notions of set theory, in particular the special set quantifiers. We show a typical *Theorema* formal text in set theory consisting of a couple of definitions and a proposition whose proof is then automatically generated by the *Theorema* set theory prover:

Definition["reflexivity", any[~ , A],
  is–reflexive$_A$[~] :⇔ $\underset{x\in A}{\forall}$ x ~ x]

Definition["symmetry", any[~ , A],
  is–symmetric$_A$[~] :⇔ $\underset{x,y\in A}{\forall}$ (x ~ y ⇒ y ~ x)]

Definition["transitivity", any[~ , A],
  is–transitive$_A$[~] :⇔ $\underset{x,y,z\in A}{\forall}$ (x ~ y ∧ y ~ z ⇒ x ~ z)]

Definition["equivalence", any[~ , A],
  is–equivalence$_A$[~] :⇔ $\bigwedge\begin{cases}\text{is–reflexive}_A[\sim]\\ \text{is–symmetric}_A[\sim]\\ \text{is–transitive}_A[\sim]\end{cases}$]

Definition["class", any[x, ~ , A],
  class$_{A,\sim}$[x] := {a ∈ A | a ~ x ∧ x ∈ A}]

Proposition["equal classes", any[x ∈ A, y ∈ A, ~ , A], with[is–equivalence$_A$[~]],
  x ~ y ⇒ (class$_{A,\sim}$[x] = class$_{A,\sim}$[y])]

Prove[Proposition["equal classes"],
  using → ⟨Definition["equivalence"], Definition["transitivity"], Definition["symmetry"], Definition["class"]⟩,
  by → SetTheoryPCSProver,
  transformBy → ProofSimplifier, TransformerOptions → {branches → Proved, steps → Useful}, ShowOptions → {},
  ProverOptions → {GRWTarget → {"goal", "kb"}, AllowIntroduceQuantifiers → True,
    UseCyclicRules → True, DisableProver → {STC, PND}, ApplyBuiltIns → {}}, SearchDepth → 50]

This call generates the following proof completely automatically. Note that, typically, in addition to the three essential arguments (proof goal, knowledge base, and prover) of the *Theorema* Prove function, the user may provide optional information that gives strategic help to the prover, determines the appearance of the output, etc.

Prove:

(Proposition (equal classes))

$$\underset{A,x,y,\sim}{\forall}(x \in A \bigwedge y \in A \bigwedge \text{is–equivalence}_A[\sim] \Rightarrow (x \sim y \Rightarrow (\text{class}_{A,\sim}[x] = \text{class}_{A,\sim}[y]))),$$

under the assumptions:

(Definition (equivalence))

$$\underset{A,\sim}{\forall}(\text{is–equivalence}_A[\sim] :\Leftrightarrow \text{is–reflexive}_A[\sim] \bigwedge \text{is–symmetric}_A[\sim] \bigwedge \text{is–transitive}_A[\sim]),$$

(Definition (transitivity))

$$\underset{A,\sim}{\forall}\left(\text{is–transitive}_A[\sim] :\Leftrightarrow \underset{x,y,z}{\forall}(x \in A \wedge y \in A \wedge z \in A \Rightarrow (x \sim y \wedge y \sim z \Rightarrow x \sim z))\right),$$

(Definition (symmetry)) $\underset{A,\sim}{\forall}\left(\text{is–symmetric}_A[\sim] :\Leftrightarrow \underset{x,y}{\forall}(x \in A \wedge y \in A \Rightarrow (x \sim y \Rightarrow y \sim x))\right),$

(Definition (class)) $\underset{A,x,\sim}{\forall}\left(\text{class}_{A,\sim}[x] := \left\{a \underset{a}{\mid} (a \sim x \wedge x \in A) \wedge a \in A\right\}\right).$

We assume

(1) $x_0 \in A_0 \bigwedge y_0 \in A_0 \bigwedge \text{is–equivalence}_{A_0}[\sim_0],$

and show

(2) $x_0 \sim_0 y_0 \Rightarrow (\text{class}_{A_0,\sim_0}[x_0] = \text{class}_{A_0,\sim_0}[y_0]).$

We prove (2) by the deduction rule.

We assume

(5) $x_0 \sim_0 y_0$

and show

(6) $\text{class}_{A_0,\sim_0}[x_0] = \text{class}_{A_0,\sim_0}[y_0].$

Formula (6), using (Definition (class)), is implied by:

(7) $\left\{a \underset{a}{\mid} x_0 \in A_0 \wedge a \in A_0 \wedge a \sim_0 x_0\right\} = \left\{a \underset{a}{\mid} y_0 \in A_0 \wedge a \in A_0 \wedge a \sim_0 y_0\right\}.$

We show (7) by mutual inclusion:

$\subseteq$: We assume

(8) $x_0 \in A_0 \wedge a1_0 \in A_0 \wedge a1_0 \sim_0 x_0$

and show:

(9) $y_0 \in A_0 \wedge a1_0 \in A_0 \wedge a1_0 \sim_0 y_0$.

We prove the individual conjunctive parts of (9):

Proof of (9.1) $y_0 \in A_0$:

Formula (9.1) is true because it is identical to (1.2).

Proof of (9.2) $a1_0 \in A_0$:

Formula (9.2) is true because it is identical to (8.2).

Proof of (9.3) $a1_0 \sim_0 y_0$:

Formula (1.3), by (Definition (equivalence)), implies:

$$\text{is–reflexive}_{A_0}[\ \sim_0\ ] \bigwedge \text{is–symmetric}_{A_0}[\ \sim_0\ ] \bigwedge \text{is–transitive}_{A_0}[\ \sim_0\ ],$$

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

(12)

$$\underset{x,y,z}{\forall} (x \in A_0 \wedge y \in A_0 \wedge z \in A_0 \Rightarrow (x \sim_0 y \wedge y \sim_0 z \Rightarrow x \sim_0 z)) \bigwedge \text{is–reflexive}_{A_0}[\ \sim_0\ ] \bigwedge \text{is–symmetric}_{A_0}[\ \sim_0\ ].$$

Formula (9.3), using (12.1), is implied by:

(13) $\underset{y}{\exists} (a1_0 \in A_0 \wedge y_0 \in A_0 \wedge y \in A_0 \wedge a1_0 \sim_0 y \wedge y \sim_0 y_0)$.

Now, let $y := x_0$. Thus, for proving (13) it is sufficient to prove:

(14) $a1_0 \in A_0 \wedge y_0 \in A_0 \wedge x_0 \in A_0 \wedge a1_0 \sim_0 x_0 \wedge x_0 \sim_0 y_0$.

We prove the individual conjunctive parts of (14):

Proof of (14.1) $a1_0 \in A_0$:

Formula (14.1) is true because it is identical to (8.2).

Proof of (14.2) $y_0 \in A_0$:

Formula (14.2) is true because it is identical to (1.2).

Proof of (14.3) $x_0 \in A_0$:

Formula (14.3) is true because it is identical to (8.1).

Proof of (14.4) $a1_0 \sim_0 x_0$:

Formula (14.4) is true because it is identical to (8.3).

Proof of (14.5) $x_0 \sim_0 y_0$:

Formula (14.5) is true because it is identical to (5).

$\supseteq$: Now we assume

.... *We do not print the second part of the proof (which is similar to the first part) in this paper ...*

■ Exampe: Induction Proofs and the Cascade

Now we demonstrate a special induction prover for equalities on natural numbers in the representation 0, 0^+, 0^{++}, ..., where $\square^+$ denotes the successor function. This prover, in its current version, is not very powerful because the underlying simplifier is not very strong. (A much better equatioanl simplifier is under development.) The main point of this demo will be the use of a general strategy, the "cascade", which often yields proofs by "inventing intermediate lemmata" for which a given prover would not be able to provide a proof by a single application. We start with an inductive definition of addition:

Definition["addition", any[m, n],

$$\begin{array}{ll} m + 0 = m & \text{" +0:"} \\ m + n^+ = (m + n)^+ & \text{" + .:"} \end{array}$$

]

Now let us try to prove commutativity by induction:

Proposition["commutativity of addition", any[m, n],
m + n = n + m " + = "]

```
Prove[Proposition["commutativity of addition"],
    using → 〈Definition["addition"]〉,
    by → NNEqIndProver,
    ProverOptions → {TermOrder → LeftToRight}, presentation → 100];
```

The attempt to generate a proof automatically by the induction prover 'NNEqIndProver' fails. In *Theorema*, in contrast to other theorem proving systems, we also generate the output for failing proofs because a lot can be learned from failing proofs. We do not show all details of the failing proof but only an essential portion of it:

....
....

Simplification of the lhs term:

$m_1{}^+ + 0 =$ by (Definition (addition): +0:)

$m_1{}^+ \rfloor$

Simplification of the rhs term:

$(0 + m_1)^+ \rfloor$

Hence, it is sufficient to prove:

(14) $m_1{}^+ = (0 + m_1)^+$.

It turns out that the prover is, basically, stuck at (14). A human prover would immediately guess that proving

$$\underset{m}{\forall}\ \mathrm{m} = 0 + \mathrm{m}$$

as a lemma, before proceeding with the proof of commutativity, would be a promising approach. This fundamental strategy of analyzing failing proofs and generating a general conjecture from the situation at which the prover is stuck, is implemented, in a recursive way, in *Theorema*. It is called the "cascade", which has basically two arguments: a prover and a conjecture generator (based on the analysis of failing proofs). In our case, the procedure can be started by entering

```
Prove[Proposition["commutativity of addition"],
      using → Definition["addition"],
   by → Cascade[NNEqIndProver, ConjectureGenerator], ProverOptions → {TermOrder → LeftToRight}];
```

This call results in the automated generation of the following sequence of proof attempts and, finally, a successful proof of commutativity. Note that, in the course of producing these proof attempts and the final proof, the initial knowledge base is automatically enlarged by lemmata, and in fact quite natural and interesting lemmata, which are *invented automatically* in the course of analyzing the failing proof attempts:

Prove:

(Proposition (+ =): + =) $\underset{m,n}{\forall}(m + n = n + m)$,

under the assumptions:

(Definition (addition): +0:) $\underset{m}{\forall}(m + 0 = m)$,

(Definition (addition): + .:) $\underset{m,n}{\forall}(m + n^+ = (m + n)^+)$.

.... *Proof fails and leads to the automated invention of the following lemma:*

$\underset{m}{\forall}(0 + m = m)$.

whose proof is now attempted next.

Prove:

(Proposition (19): 19) $\underset{m}{\forall}(0+m=m)$,

under the assumptions:

(Definition (addition): +0:) $\underset{m}{\forall}(m+0=m)$,

(Definition (addition): + .:) $\underset{m,n}{\forall}(m+n^{+}=(m+n)^{+})$.

.... This proof succeeds and, hence, the lemma can be added to the knowledge base and the proof of commutativity is attempted again.

Prove:

(Proposition (+ =): + =) $\underset{m,n}{\forall}(m+n=n+m)$,

under the assumptions:

(Proposition (19): 19) $\underset{m}{\forall}(0+m=m)$,

(Definition (addition): +0:) $\underset{m}{\forall}(m+0=m)$,

(Definition (addition): + .:) $\underset{m,n}{\forall}(m+n^{+}=(m+n)^{+})$.

.... Proof fails and leads to the automated invention of the following lemma:

$\underset{n,m}{\forall}(n^{+}+m=(n+m)^{+})$.

whose proof is now attempted next.

Prove:

(Proposition (15): 15) $\underset{n,m}{\forall}(n^{+}+m=(n+m)^{+})$,

under the assumptions:

(Proposition (19): 19) $\underset{m}{\forall}(0+m=m)$, (Definition (addition): +0:) $\underset{m}{\forall}(m+0=m)$,

(Definition (addition): + .:) $\underset{m,n}{\forall}(m+n^{+}=(m+n)^{+})$.

.... This proof succeeds and, hence, the lemma can be added to the knowledge base and the proof of commutativity is attempted again.

whose proof is now attempted next.

Prove:

(Proposition (+ =): + =) $\underset{m,n}{\forall}(m+n=n+m)$,

under the assumptions:

(Proposition (15): 15) $\underset{n,m}{\forall}(n^{+}+m=(n+m)^{+})$,

(Proposition (19): 19) $\underset{m}{\forall}(0+m=m)$,

(Definition (addition): +0:) $\underset{m}{\forall}(m+0=m)$,

(Definition (addition): + .:) $\underset{m,n}{\forall}(m+n^{+}=(m+n)^{+})$.

.... This proof succeeds and, hence, commutativity is proved.

The cascade can also be viewed as a kind of general completion procedure that goes far beyond the Knuth-Bendix completion procedure for equational theorem proving.

■ Example: Proving Booleans of Equalities by Groebner Bases

The next example concerns a special, but quite powerful, prover / disprover (decision algorithm) based on the author's Groebner bases method for arbitrary universally quantified boolean combinations of arithmetical equalities over complex numbers. Here is a typical formula in this class:

$$\textbf{Formula}\Big[\text{"Test"}, \text{any}[x, y], \; ((x^2 y - 3x) \neq 0) \vee ((x y + x + y) \neq 0) \vee (((x^2 y + 3x = 0) \vee ((-2x^2 + -7xy + x^2 y + x^3 y + -2y^2 + -2xy^2 + 2x^2 y^2) = 0)) \wedge ((x^2 + -xy + x^2 y + -2y^2 + -2xy^2) = 0)) \quad \text{"B1"}\Big]$$

The following *Theorema* call produces, fully automatically a proof that this formula is true over the complex numbers:

```
Prove[Formula["Test"], using → {}, by → GroebnerBasesProver]
```

- ProofObject -

Here is the automatically generated proof text:

Prove:

(Formula (Test): B1)

$$\forall_{x,y} (((x^2 * y - 3 * x) \neq 0) \bigvee (x * y + x + y \neq 0) \bigvee ((x^2 * y + 3 * x = 0) \bigvee ((-2) * x^2 + (-7) * x * y + x^2 * y + x^3 * y + (-2) * y^2 + (-2) * x * y^2 + 2 * x^2 * y^2 = 0)) \bigwedge (x^2 + (-x) * y + x^2 * y + (-2) * y^2 + (-2) * x * y^2 = 0)) ,$$

with no assumptions.

Proved.

The Theorem is proved by the Groebner Bases method.

The formula in the scope of the universal quantifier is transformed into an equivalent formula that is a conjunction of disjunctions of equalities and negated equalities. The universal quantifier can then be distributed over the individual parts of the conjunction. By this, we obtain:

Independent proof problems:

(Formula (Test): B1.1)

$$\forall_{x,y} ((x^2 + (-x * y) + x^2 * y + (-2) * y^2 + (-2) * x * y^2 = 0) \bigvee ((-3) * x + x^2 * y \neq 0) \bigvee (x + y + x * y \neq 0))$$

(Formula (Test): B1.2)

$$\forall_{x,y} ((3 * x + x^2 * y = 0) \bigvee ((-2) * x^2 + (-7) * x * y + x^2 * y + x^3 * y + (-2) * y^2 + (-2) * x * y^2 + 2 * x^2 * y^2 = 0) \bigvee ((-3) * x + x^2 * y \neq 0) \bigvee (x + y + x * y \neq 0))$$

We now prove the above individual problems separately:

Proof of (Formula (Test): B1.1):

... We do not print this part of the proof in this paper, since it is easier than the second part of the proof. ...

Proof of (Formula (Test): B1.2):

This proof problem has the following structure:

(Formula (Test): B1.2.structure) $\underset{x,y}{\forall}((\text{Poly}[1] \neq 0) \vee (\text{Poly}[2] \neq 0) \vee (\text{Poly}[3] = 0) \vee (\text{Poly}[4] = 0))$,

where

$\text{Poly}[1] = (-3) * x + x^2 * y$
$\text{Poly}[2] = x + y + x * y$
$\text{Poly}[3] = 3 * x + x^2 * y$
$\text{Poly}[4] = (-2) * x^2 + (-7) * x * y + x^2 * y + x^3 * y + (-2) * y^2 + (-2) * x * y^2 + 2 * x^2 * y^2$

(Formula (Test): B1.2.structure) is equivalent to

(Formula (Test): B1.2.implication) $\underset{x,y}{\forall}((\text{Poly}[1] = 0) \wedge (\text{Poly}[2] = 0) \Rightarrow (\text{Poly}[3] = 0) \vee (\text{Poly}[4] = 0))$

.

(Formula (Test): B1.2.implication) is equivalent to

(Formula (Test): B1.2.not-exists)

$\underset{x,y}{\nexists}(((\text{Poly}[1] = 0) \wedge (\text{Poly}[2] = 0)) \wedge ((\text{Poly}[3] \neq 0) \wedge (\text{Poly}[4] \neq 0)))$.

By introducing the slack variable(s)

$\{\xi 1, \xi 2\}$

(Formula (Test): B1.2.not-exists) is transformed into the equivalent formula

(Formula (Test): B1.2.not-exists-slack)

$\underset{x,y,\xi 1,\xi 2}{\nexists}((\text{Poly}[1] = 0) \wedge (\text{Poly}[2] = 0) \wedge (-1 + \xi 1\, \text{Poly}[3] = 0) \wedge (-1 + \xi 2\, \text{Poly}[4] = 0))$.

Hence, we see that the proof problem is transformed into the question on whether or not a system of polynomial equations has a solution or not. This question can be answered by checking whether or not the (reduced) Groebner basis of

$\{\text{Poly}[1], \text{Poly}[2], -1 + \xi 1\, \text{Poly}[3], -1 + \xi 2\, \text{Poly}[4]\}$

is exactly $\{1\}$.

Hence, we compute the Groebner basis for the following polynomial list:

$\{-1 + 3\, x\, \xi 1 + x^2\, y\, \xi 1,$
$-1 - 2\, x^2\, \xi 2 - 7\, x\, y\, \xi 2 + x^2\, y\, \xi 2 + x^3\, y\, \xi 2 - 2\, y^2\, \xi 2 - 2\, x\, y^2\, \xi 2 + 2\, x^2\, y^2\, \xi 2, -3\, x + x^2\, y, x + y + x\, y\}$

The Groebner basis:

$\{1\}$

Hence, (Formula (Test): B1.2) is proved.

Since all of the individual subtheorems are proved, the original formula is proved.

□

In fact, the proof text produced does not only show the essential steps in the proof but also explains the proof method itself in every particular example proof: It shows, at the beginning of the proof, how the initial proof problem can be reduced to the problem of computing certain Groebner bases.

■ Example: Inventing and Proving Combinatorial Identities Involving the Sum Quantifier

Recently, much progress has been made on the invention and proof of combinatorial identities, i.e identities involving combinatorial functions and the summation and product quantifiers. Some of the methods used in this area are also based on a non-commutative version of the Groebner bases method. In *Theorema*, the work of the research group of Peter Paule at RISC is also integrated. The respective prover can be called under the name 'Paule-Schorn–Telescope' method. For example, for the following sum

Formula["SIAM series",

$$\sum_{k=1,\ldots,n} \frac{(-1)^{k+1} (4\,k+1)\,(2\,k)!}{2^k\,(2\,k-1)\,(k+1)!\,2^k\,k!}]$$

the call

Prove[Formula["SIAM series"], by → PauleSchorn–Telescope, built–in → Built–in["PauleSchorn"]]

automatically invents the following closed form and provides the following proof for its correctness:

Theorem:

If $-1+n$ is a natural number, then:

$$\sum_{k=1,\ldots,n} \left(\frac{-(-1)^k\, 2^{-2\,k}\,(2\,k)!\,(1+4\,k)}{k!\,(1+k)!\,(-1+2\,k)} \right) = 1 + \frac{-(-1)^n\, 2^{-2\,n}\,(2\,n)!}{n!\,(1+n)!}$$

Proof:

Let Δ_k denote the forward difference operator in k then

the Theorem follows from summing the equation

$$\frac{-(-1)^k\, 2^{-2\,k}\,(2\,k)!\,(1+4\,k)}{k!\,(1+k)!\,(-1+2\,k)} = \Delta_k\left[\frac{(-1)^k\, 2^{1+-2\,k}\,(2\,k)!\,(1+k)}{k!\,(1+k)!\,(-1+2\,k)}\right]$$

over the range $k = 1, \ldots, n$.

The equation is routinely verifiable
by dividing the right–hand side by the left–hand side
and simplifying the resulting rational function

$$\frac{\frac{(-1)^{1+k}\, 2^{1+-2\,(1+k)}\,(2\,(1+k))!\,(2+k)}{(1+k)!\,(2+k)!\,(-1+2\,(1+k))} - \frac{(-1)^{k}\, 2^{1+-2\,k}\,(2\,k)!\,(1+k)}{k!\,(1+k)!\,(-1+2\,k)}}{\frac{-(-1)^{k}\, 2^{-2\,k}\,(2\,k)!\,(1+4\,k)}{k!\,(1+k)!\,(-1+2\,k)}}$$

to 1.

The proof of the (automatically invented) theorem is, hence, reduced to a test whether a certain expression can be simplified to 1. This test can be executed by calling the Mathematica algorithm 'FullSimplify' (which, of course, could also be built into the above *Theorema* prover as a last step):

$$\text{System`FullSimplify}\left[\frac{\frac{(-1)^{1+k}\, 2^{1+-2\,(1+k)}\,(2\,(1+k))!\,(2+k)}{(1+k)!\,(2+k)!\,(-1+2\,(1+k))} - \frac{(-1)^{k}\, 2^{1+-2\,k}\,(2\,k)!\,(1+k)}{k!\,(1+k)!\,(-1+2\,k)}}{\frac{-(-1)^{k}\, 2^{-2\,k}\,(2\,k)!\,(1+4\,k)}{k!\,(1+k)!\,(-1+2\,k)}}\right]$$

1

This example demonstrates the power of this recent method: The problem of constructing a closed form for the above input expression was an open problem for many years! The solution was given in P. Paule, *Computer-Solution of Problem 94-2,* SIAM REVIEW Vol.37 (1995), 105-106 using the Paule-Schorn automated conjecture generator / prover.

■ Example: Calling External Provers (e.g. Resolution)

We have also provide links from *Theorema* to various existing theorem provers, for example to the Otter prover for predicate logic which is based on the resolution method. For example, after entering the definitions

Definition["Inclusion",

$\underset{A,B}{\forall}\left((A \subseteq B) \Longleftrightarrow \underset{x}{\forall}((x \in A) \Longrightarrow (x \in B))\right)$ "⊆:"]

Definition["Union",

$\underset{A,x}{\forall}\left((x \in \bigcup A) \Longleftrightarrow \underset{Y}{\exists}((x \in Y) \wedge (Y \in A))\right)$ "⋃:"]

Definition["Powerset",

$\underset{A,x}{\forall}((x \in \mathcal{P}[A]) \Longleftrightarrow (x \subseteq A))$ "P:"]

we can generated an Otter proof of the following proposition

Proposition["Union of powerset is included...",

$\underset{A}{\forall}(\bigcup \mathcal{P}[A] \subseteq A)$ "P⋃⊆"]

by the *Theorema* call

ProveExternal[Proposition["Union of powerset is included..."],
using → {**Definition**["Inclusion"], Definition["Union"], Definition["Powerset"]}, by → Otter]

On termination, various information on the resolution proof is returned. The most essential part is the sequence of resolution steps that lead to the empty clause. In our case, this sequence has the following form:

```
1[]-subsetequal(union(powerset($c1)),$c1).
2[]-subsetequal(A,B)|-element(C,A)|element(C,B).
3[] subsetequal(A,B)|-element($f1(A,B),B).
4[]-element(A,union(B))|element(A,$f2(B,A)).
5[]-element(A,union(B))|element($f2(B,A),B).
7[]-element(A,powerset(B))|subsetequal(A,B).
10[] subsetequal(A,B)|element($f1(A,B),A).
12[hyper,10,1] element($f1(union(powerset($c1)),$c1),union(powerset($c1))).
35[hyper,12,5] element($f2(powerset($c1),$f1(union(powerset($c1)),$c1)),powerset($c1)).
36[hyper,12,4] element($f1(union(powerset($c1)),$c1),$f2(powerset($c1),$f1(union(powerset($c1)),$c1))).
1376[hyper,35,7] subsetequal($f2(powerset($c1),$f1(union(powerset($c1)),$c1)),$c1).
5347[hyper,36,2,1376] element($f1(union(powerset($c1)),$c1),$c1).
5407[hyper,5347,3] subsetequal(union(powerset($c1)),$c1).
5408[binary,5407.1,1.1] $F.
```

We see that, in fact, several thousand intermediate clauses were produced during the resolution proof. Of course, the resulting proof is not meant to be "understood" or "read" by humans. This is in sharp contrast to the provers which we try to produce as internal provers of the *Theorema* system whose emphasis is on readability and naturalness.

■ Example: Computation Using Functors

Finally, we want to illustrate that *Theorema* smoothly integrates the entire computational power of Mathematica and, in fact, goes beyond Mathematica as a computing engine in two important respects: We provide the functor construct and we allow the explict indication of knowledge bases w.r.t. which a computation (i.e. iterated simplification) should be executed. The *Theorema* functor construct, which is similar to but more general than the functor construct in ML, allows to define processes by which new domains (carriers together with functions and predicates) are produced from arbitrary given domains. As an example, we show the functor 'pol' that takes a domain C of coefficients and a domain T of terms (power products) and constructs the domain P of polynomials over C and T represented as tuples of pairs of coefficients and power products ordered by the ordering available on the terms.

Definition["pol", any[C, T],

pol[C, T] = Functor[P, any[c, d, i, $\overline{m}$, $\overline{n}$, p, q, s, t],

$$\frac{\mathbb{s} = \langle\rangle}{}$$

$\underset{P}{0} = \langle\rangle$

$\underset{P}{1} = \left\langle\left\langle \underset{C}{1}, \underset{T}{1} \right\rangle\right\rangle$

$\langle\rangle \underset{P}{+} q = q$

$p \underset{P}{+} \langle\rangle = p$

$\langle\langle c, s\rangle, \overline{m}\rangle \underset{P}{+} \langle\langle d, t\rangle, \overline{n}\rangle = \left\| \langle c, s\rangle \smile \left(\langle\overline{m}\rangle \underset{P}{+} \langle\langle d, t\rangle, \overline{n}\rangle\right) \Leftarrow s \underset{T}{>} t, \right.$

$\quad \langle d, t\rangle \smile \left(\langle\langle c, s\rangle, \overline{m}\rangle \underset{P}{+} \langle\overline{n}\rangle\right) \Leftarrow t \underset{T}{>} s,$

$\quad \left(\left\langle c \underset{C}{+} d, s\right\rangle\right) \smile \left(\langle\overline{m}\rangle \underset{P}{+} \langle\overline{n}\rangle\right) \Leftarrow \left(c \neq \underset{C}{-} d\right),$

$\left.\left(\langle\overline{m}\rangle \underset{P}{+} \langle\overline{n}\rangle\right) \Leftarrow \text{otherwise} \right\|$

$\underset{P}{-} \langle\rangle = \langle\rangle$

$\underset{P}{-} \langle\langle c, s\rangle, \overline{m}\rangle = \left\langle \underset{C}{-} c, s\right\rangle \smile \left(\underset{P}{-} \langle\overline{m}\rangle\right)$

$p \underset{P}{-} q = p \underset{P}{+} \left(\underset{P}{-} q\right)$

$\langle\rangle \underset{P}{*} q = \langle\rangle$

$p \underset{P}{*} \langle\rangle = \langle\rangle$

$\langle\langle c, s\rangle, \overline{m}\rangle \underset{P}{*} \langle\langle d, t\rangle, \overline{n}\rangle = \left(\left\langle\left\langle c \underset{C}{*} d, s \underset{T}{*} t\right\rangle\right\rangle \underset{P}{+} \langle\langle c, s\rangle\rangle \underset{P}{*} \langle\overline{n}\rangle\right) \underset{P}{+} \langle\overline{m}\rangle \underset{P}{*} \langle\langle d, t\rangle, \overline{n}\rangle$

]]

Having introduced this functor, one can now apply it to particular coefficient and power product domains and produce, by one functor call, the corresponding polynomial domain in tuple representation. Let us assume, for example, that the two domains $\mathbb{Z}$ (integers) and $\mathbb{T}$ (trivariate power products as triples of natural numbers encoding the exponents at the three indeterminates) have already been defined (using appropriate parameter-less functors) in appropriate definitions Definition["integers"] and Definition["power products"]. Building up the corresponding polynomial domain $\mathbb{P}$ and computing in $\mathbb{P}$ would then proceed as follows:

Theory["F",

Definition["integers"]
Definition["power products"]]
Definition["pol"]

Definition["D",

$\mathbb{P} = \text{pol}[\mathbb{Z}, \mathbb{T}]$]

Use[⟨Built–in["Quantifiers"], Built–in["Connectives"], Built–in["Numbers"],
Built–in["Tuples"], Built–in["Sets"], Built–in["natural numbers"], Theory["F"], Definition["D"]⟩]

Note that, in addition to user–defined knowledge like Theory["F"] and Definition["D"], in *Theorema* one can also explicitly make available built-in computational knowledge (algorithms) from the underlying Mathematica and *Theorema* library by using the 'Built–in' construct. Now, for example, the call

$$\text{Compute}\left[\left(\langle\langle 5, \langle 2, 3, 1\rangle\rangle, \langle 3, \langle 1, 6, 3\rangle\rangle\rangle \underset{\mathbb{P}}{*} \left(\langle\langle 5, \langle 2, 3, 1\rangle\rangle, \langle 3, \langle 1, 6, 3\rangle\rangle\rangle \underset{\mathbb{P}}{*} \langle\langle 5, \langle 2, 3, 1\rangle\rangle, \langle 3, \langle 1, 6, 3\rangle\rangle\rangle\right)\right)\right]$$

produces

$$\langle\langle 125, \langle 6, 9, 3\rangle\rangle, \langle 225, \langle 5, 12, 5\rangle\rangle, \langle 135, \langle 4, 15, 7\rangle\rangle, \langle 27, \langle 3, 18, 9\rangle\rangle\rangle$$

which, in the usual representation of trivariate polyonomials, is the result $125\, x1^6\, x2^9\, x3^3 + 225\, x1^5\, x2^{12}\, x3^5 + 135\, x1^4\, x2^{15}\, x3^7 + 27\, x1^3\, x2^{18}\, x3^9$ of expanding $(5\, x1^2\, x2^3\, x3 + 3\, x1\, x2^6\, x3^3)^3$.

■ Conclusion

I believe that, going into the direction of systems like *Theorema*, the following will soon be possible:

- Computer-support of all aspects of doing mathematics will reach higher and higher levels including inventing, exploring, proving, and managing mathematical knowledge.

- All aspects of doing mathematics are supportable in one common logical and software-technological frame.

- In particular, nonalgorithmic and algorithmic mathematics are supportable in one common logical and software-technological frame (in other words, mathematics, and computer science will reconcile).

As a result, the way how we invent, teach, and apply mathematics and the way how mathematical knowledge is published, stored in knowledge bases and retrieved will change drastically. For this change to happen, the improvement of mathematical systems along the lines of systems like *Theorema* is one important prerequisite but on the only one. The other one is that the way how the next generation of math students is trained must drastically change, namely into the direction of giving them much more formal training and culture.