The TH∃OREM∀ Project: A Progress Report *

B. Buchberger, C. Dupré, T. Jebelean, F. Kriftner, K. Nakagawa, D. Văsaru, W. Windsteiger

Research Institute for Symbolic Computation
A-4232 Schloß Hagenberg, Austria

firstname.lastname@risc.uni-linz.ac.at

Abstract

The TH∃OREM∀ project aims at supporting, within one consistent logic and one coherent software system, the entire mathematical exploration cycle including the phase of proving. In this paper we report on some of the new features of TH∃OREM∀ that have been designed and implemented since the first expository version of TH∃OREM∀ in 1997. These features are:

- the TH∃OREM∀ formal text language
- the TH∃OREM∀ computational sessions
- the Prove-Compute-Solve (PCS) prover of TH∃OREM∀
- the TH∃OREM∀ set theory prover
- special provers within TH∃OREM∀
- the cascade-meta-strategy for TH∃OREM∀ provers
- proof simplification in TH∃OREM∀.

In the conclusion, we formulate design goals for the next version of TH∃OREM∀.

1 Introduction

In [8] we described the objectives and the design of the TH∃OREM∀ system and the state of the implementation in 1997. In the present paper, we report on the progress made in the TH∃OREM∀ project since 1997.

The TH∃OREM∀ project aims at supporting, within one consistent logic and one coherent software system, the entire mathematical exploration cycle consisting of

- introducing new notions (functions, predicates, algorithms) by definitions or axioms
- conjecturing and proving / disproving mathematical facts about the new notions w.r.t. knowledge bases consisting of definitions, axioms, lemmata, theorems etc.
- extracting methods and algorithms from proved facts
- applying algorithms to input data.

Hence, in a simplified view, TH∃OREM∀ tries to combine the functionality of present mathematical software systems and theorem proving systems. Worldwide, there are quite some projects under way that aim at combining the power of these two classes of systems. Notably, the research groups integrated in the EU Calculemus Consortium share the view that this combination is essential for the future development of computer-supported mathematics and its applications. The projects of the Calculemus groups are described in [1], [3], [4], [9], [10], [19], [21]. Other projects with similar objectivess are described in [2], [5], [12], [13], [14], [18].

The objectives and features that distinguish the TH∃OREM∀ project from the other, similar, projects are the following:

- We emphasize "theory exploration" over "theorem proving", i.e. the goal of TH∃OREM∀ is to support the user in formally developing and proving big portions of coherent mathematical texts (like lecture notes and textbooks) rather than in proving isolated theorems, see [7].
- We consider both computer-support in the "routine" reductions of proof problems to known black-box techniques and also the computer-supported invention of "ingenious" proofs by strong special techniques (e.g. algebraic techniques) based on nontrivial mathematics.
- We emphasize automated proof generation over automated proof checking.
- We emphasize, in automated routine proofs, the importance of generating human-readable proof presentations in a "natural style".
- We integrate proving, solving, and simplifying, as the main three formal activities in exploring mathematics, in one coherent system.
- We integrate many different general and special provers, solvers, and simplifiers into one system, i.e. TH∃OREM∀ is a multi-method system.
- We use a commercial mathematical software system (*Mathematica* [22]) as our programming environment so that TH∃OREM∀ is available on all platforms.

*This research is partially supported by the Austrian Science Foundation (FWF-SFB-project FO-1302) and the Upper Austrian Government (project "PROVE").

- We emphasize the importance of convenient, two-dimensional, extensible syntax and graphical illustration.

The new features of TH∃OREM∀ on which we report in the present paper are:

- the TH∃OREM∀ formal text language
- the distinction between TH∃OREM∀ standard sessions and TH∃OREM∀ computational sessions
- a new predicate logic prover that integrates proving, simplifying and solving steps in a systematic way
- a set theory prover that smoothly fits into the philosophy of this predicate logic prover
- the integration of powerful special provers as, for example, the Groebner bases prover and the Gosper-Zeilberger prover [17]
- tools for extending existing provers by general strategies
- tools for proof simplification.

2 The TH∃OREM∀ Formal Text Language

The core language (expression language) of TH∃OREM∀ ([8]) is a version of higher order predicate logic. Expressions like

$$\langle X_i + Y_i \mid_{i=1,\ldots,|X|} \rangle$$

$$\forall_{a,b,c\in A} \; (a\tilde{b} \wedge b\tilde{c} \Rightarrow a\tilde{c})$$

$$\forall_{\epsilon>0} \; \exists_{\delta>0} \; \forall_{|y-x|<\delta} \; |f[y] - f[x]| < \epsilon$$

are examples of terms and formulae in this language.

However, for composing and manipulating large formal mathematical texts as for example reports, publications, lecture notes, and monographs, we need to be able to combine the expression language with auxiliary text (labels, key words like "Definition", "Proposition", "Theorem", etc.) and to compose, in a hierarchical way, large mathematical knowledge bases from individual expressions. For this, we designed and implemented the "TH∃OREM∀ Formal Text Language".

Here are some typical examples of formal text written in the TH∃OREM∀ formal text language:

Definition["continuity", any$[f, x]$,
continuous$[f, x] :\Leftrightarrow \forall_{\epsilon>0} \exists_{\delta>0} \forall_{|y-x|<\delta} |f[y] - f[x]| < \epsilon$ "c2:"]

Definition["fprod", any$[f, g, x]$,
$(f * g)[x] = f[x] * g[x]$ "f*g"]

Proposition["continuity of product", any$[f, x]$,
continuous$[f, x] \wedge$ continuous$[g, x] \Rightarrow$ continuous$[f * g, x]$ "cont*"]

Finally, in order to process knowledge, we provide the "TH∃OREM∀ Command Language" in order to *prove* propositions, *compute* values (i.e. *simplify* terms or formulae), or *solve* problems, see also Section 2.3.

In the sequel, we will describe the most important features of the "TH∃OREM∀ Formal Text Language". The Formal Text Language contains 3 categories of Formal Text Elements:

Environments for organizing knowledge in definitions, propositions, theorems, etc.,

Built-ins for assigning a "built-in interpretation" to certain symbols, and

Properties for asserting properties of certain operators.

2.1 TH∃OREM∀ Environments

TH∃OREM∀ environments allow to enter knowledge into the system in a style similar to how definitions, propositions etc. are given in mathematical textbooks. Consider a definition like

Definition 1 (Sum of Tuples) *For any two tuples X and Y with $|X| = |Y|$ we define*

$$X \oplus Y := \langle X_i + Y_i \mid_{i=1,\ldots,|X|} \rangle \tag{1}$$

The ingredients of a structure like this are a keyword ("Definition", "Lemma", "Theory", etc. .), a label for later reference ("Sum of Tuples"), one or more mathematical statement(s) (a formula/term or a defintion of a new function/predicate), an enumeration of the (free) variables occurring in the statement (X and Y), and, if necessary, conditions on the variables or relations among them. The TH∃OREM∀ Formal Text Language supports input of the above definition in the following format:

Definition["Sum of Tuples", any$[X, Y]$, with$[|X| = |Y|]$
$X \oplus Y := \langle X_i + Y_i \mid_{i=1,\ldots,|X|} \rangle$ "$\langle\rangle + \langle\rangle$"]

More abstract, an environment has the form

Keyword[*env_label*,{any[*vars*] {,with[*cond*]}, }
clause$_1$ {*label*$_1$} | *keyword*$_1$[*env_label*$_1$]
{*more clauses*} | {*more references*}]

where all fields enclosed in {} are optional and the | denotes an alternative. The user is free to choose any string for *env_label* and the clause labels *label*$_i$. Omitting the clause labels assigns "1", "2", etc. automatically. Labels do not carry any semantics, they are only used for referring to environments and formulae in proofs and computations. The field "any[*vars*]" declares *vars* as (the free) variables. Each variable v in *vars* can carry a type restriction of the form "type[v]" (see also the example in Section 3). The field "with[*cond*]" tells that the variables must satisfy the condition *cond*.

The effect of entering an environment into the system is that the environment is transformed into an internal representation that can be referred to later by *Keyword*[*env_label*]. Knowledge can be grouped using *nested environments*, whose structure is identical except that instead of clauses (formulae with optional labels) there are references to previously defined environments. Typical keywords used for nested environments are "Theory" and "KnowledgeBase".

2.2 Built-ins and Properties

TH∃OREM∀ gives the user full control over the interpretation of any symbol, hence, the automatic interpretation of symbols by the underlying *Mathematica* system must be avoided. For this, the user has the possibility to give implicit knowledge about the interpretation of symbols using the Formal Text Element "**Built-in**". Entering

Built-in["My ops",
$+ \rightarrow$ Times
$* \rightarrow$ Plus]

defines Built-in["My ops"] to translate "+" into the the *Mathematica* built-in function `Times` and "*" into `Plus`. In addition, we provide various translations of symbols to their "usual" meaning. Using the Formal Text Element "**Property**" in a similar fashion, it is possible to assert properties of operators (e.g. commutativity of "+"). Each TH∃OREM∀ command then provides the possibility to obey implicit knowledge of that kind.

2.3 The TH∃OREM∀ Command Language

In a TH∃OREM∀ standard session, the user has maximum control over processing knowledge, i.e. the user can for instance specify an explicit knowledge base, implicit knowledge about operators, or the appropriate prove (simplify, solve) method. Typical TH∃OREM∀ commands are:

`Prove`[Proposition["continuity of product"],
using→ {Definition["continuity"], Definition["fprod"]},
by→PCS],

`Compute`[$\langle 1,2,3\rangle \oplus \langle 7,1,-3\rangle$,
using→ {Definition["Sum of Tuples"],
built-in→ {Built-in["Operators"], Built-in["Tuples"]}], or

`Compute`[$\langle 1,2,3\rangle \oplus \langle 7,1,-3\rangle$,
using→ {Definition["Sum of Tuples"],
built-in→ {Built-in["My ops"], Built-in["Tuples"]}],

where the options have the following meaning:

using defines the explicit knowledge base to be used.

built-in defines implicit knowledge about symbols used.

by specifies the method to be applied.

Note the difference in the last two computations: The first one uses "normal" interpretation of symbols provided in Built-in["Operators"] and, thus, results in $\langle 8,3,0\rangle$, whereas the latter employs the user-defined interpretation Built-in["My ops"] from the previous section, thus resulting in $\langle 7,2,-9\rangle$.

Moreover, the command language contains administrative commands in order to adjust global settings guarding the system's behavior and to maintain global values. Global settings can be given through the command `SetGlobals`, its most frequent use is

`SetGlobals`[Evaluator→*e_method*, Prover→*p_method*] for defining *e_method* as the preferred evaluation method and *p_method* as the preferred prove method.

The commands `Prove` and `Compute` are aware of globally given explicit and implicit knowledge bases, which are adjoined to the knowledge passed through the options `using` and `built-in`. Global knowledge can be given or removed by the commands `Use` and `DoNotUse`, respectively. After

`Use`[{Definition["continuity"], Definition["fprod"]}] the Prove-call above can be reduced to

`Prove`[Proposition["continuity of product"]].

3 The TH∃OREM∀ Computational Session

As we saw in the example above, in a TH∃OREM∀ standard session, the user interacts with TH∃OREM∀ by, first, specifying various definitions, axioms, propositions, and knowledge bases built up from such entities and, then, calls a TH∃OREM∀ prover, simplifier, or solver using the TH∃OREM∀ Command Language.

The explicit indication of the knowledge base used and the prove (simplifiy, solve) method applied gives maximum control over the formal development of a mathematical text. This is an important feature of TH∃OREM∀. Typically, current mathematical software systems lack this feature, which is the reason why logically important side-conditions (like conditions on parameters in integrals etc.) cannot be modelled correctly in these systems, see however recent extensions to mathematical software systems like the `Assumptions` option to some commands in *Mathematica* 4 or the `assume` facility in MAPLE.

However, often, the knowledge base used and the method applied is fixed for a long part of a formal text (for example, for an entire section of a book). For such situations, TH∃OREM∀ now provides two facilities: One can either define the knowledge base and/or the prove method applied as a global parameter or one can switch to a "computational session". In such a session, it is tacitly assummed that every new definition, axiom, proposition etc. is added to the global knowledge base and that a standard simplifier is applied to the expression entered into an input cell. In other words, in a computational session, TH∃OREM∀ behaves very much the same as *Mathematica* or any of the other mathematical software systems. Moreover, the computational session does not need environments as described in Section 2, since there is neither need to refer to individual formulae nor to group them into nested structures.

In order to switch to a computational session, use the command

`ComputationalSession`[using→
Definition["Sum of Tuples"]]

The option `using` gives the opportunity to import knowledge available in the standard session into the global knowledge base for the computational session. Alternatively, one can import this definition using the command

`Use`[Definition["Sum of Tuples"]]

from inside a computational session. Symbols defined in the standard session are invisible in the computational session unless they are imported. Instead of entering knowledge through environments, definitions can be given directly to the system:

any[is-set[A,B]]:
$$A \ominus B := \{x \mid_{x \in A} x \notin B\}$$

In general, a definition has the form

{any[*vars*]{,with[*cond*]}}:

$lhs := rhs$

{ *more definitions* }

where "any[...]" and "with[...]" have the same semantics like inside an environment in the standard session (note the type specification in the example!). Unlike in a standard session, one can now simply type

$$\{2, 3, 1, 3\} \ominus \{3, 5\}$$

into an input cell of *Mathematica* and evaluate to $\{1, 2\}$ without having to (without being able to!) specify any knowledge base or evaluation method. The knowledge base is the accumulated knowledge built up during the current session, the evaluation method is *Mathematica*'s default expression evaluator. There is no possibility to give implicit knowledge about operator symbols.

4 The TH∃OREM∀ Prove-Simplify-Solve Prover for Predicate Logic

Many interesting mathematical notions are defined by formulae whose syntactical structure is characterized by a sequence of "alternating quantifiers", i.e. the definitions have the structure

$$p[x, y] \Leftrightarrow \underset{a}{\forall}\, \underset{b}{\exists}\, \underset{c}{\forall} \ldots q[x, y, a, b, c, \ldots]$$

The exploration of theories about notions introduced by such definitions aims at proving, first of all, an arsenal of "rewrite rules" for these notions which later will be helpful in the subsequent proofs of more complicated theorems or for the purpose of "computing examples" involving these notions.

For example, most of the notions introduced in elementary analysis text books (like the notion of limit, the notion of continuity, the notion of the growth order of a funtion etc.) fall into this class. The automated proof of propositions about such notions is, therefore, a practically important challenge for future mathematical systems, as was pointed out in [6]. Meanwhile, this class of propositions is used by various people as a test set for their systems, see for example [15].

In TH∃OREM∀ , we now implemented a new prover for predicate logic, which we call the "PCS" ("Prove, Compute, Solve") prover that is particularly suited for proving theorems about notions of the above kind, which, in this paper, we call "alternating quantifiers theorems". The basic strategy of this prover, which simulates what we believe is a frequent and natural strategy used by human provers for routine proofs about alternating quantifiers theorems, is as follows:

- *"Preparation Phase"*: In the knowledge base of a proof situation (for basic concepts like "proof situation", see [8]), we distinguish between "rewrite-formulae" and "non-rewrite-formulae". The rewrite-formulae are basically all equalities, equivalences and implications (of the form $A \Leftarrow B$) whose left-hand side does not contain quantifiers nor propositional connectives.

  Also, in the knowledge base, all formulae

  $$\underset{P[x]}{\overset{x}{\forall}}\, \underset{Q[y]}{\overset{y}{\exists}}\, F[x, y]$$

  are transformed into

  $$\underset{P[x]}{\overset{x}{\forall}}\, Q[S[x]] \wedge F[x, S[x]]$$

  by introducing a Skolem function S. (Similarly, for formulae with more alternations of quantifiers).

- *"Prove Phase"*: We first apply all the usual inference rules (in our particular version of natural deduction, see [8]) of propositional and predicate logic to the goal and the non-rewrite formulae in the knowledge base until no more such rule can be applied.

- *"Compute Phase"*: Now we use all the rewrite-formulae for simplifying the goal and the formulae of the knowledge base ("computing"). Note that rewrite rules that stem from an implication "expand the knowledge" but "reduce the goal". Note also that, in this phase, we allow "semantical pattern matching" (see next paragraph), which is much stronger than ordinary syntactical pattern matching. The compute phase may introduce new formulae in the knowledge base or new goals that can be manipulated as described in the preparation phase and, also, allow again the application of rules of the prove phase. Hence, we may need to circle through the preparation, prove and compute phases a couple of times (which, in the exceptional case, may already yield a proof) before we enter the next phase.

- *"Solve Phase"*: Now we are left with a proof situation in which the goal has the form

  $$\underset{x,y}{\exists}\, G[x, y, \ldots].$$

  This situation essentially specifies a "find problem": We have to find x^*, y^*, such that $G[x^*, y^*, \ldots]$ is true under the assumptions collected in the knowledge base. In this phase, a couple of general rules for transforming the solve problem are applied and then, depending on the type of the variables $x, y, \ldots$, special solvers are called. For example, if $x, y, \ldots$ are variables ranging over real numbers we call (a full or specialized) version of Collins' cad-method (see [11]) (which is implemented in *Mathematica* version 4.0, see [20]).

"Semantical pattern matching": We explain this idea in an example. Assume that, in the knowledge base, we have the formula

$$\underset{x,y,z,t,\delta,\epsilon}{\forall} (|y * t - x * z| < \delta * (\epsilon + |z| + 1) + |x| * \epsilon \Leftarrow (|y - x| < \delta \wedge |t - z| < \epsilon))$$

and the proof goal contains

$$|f_0[y] * g_0[y] - f_0[x_0] * g_0[x_0]| < \epsilon_0.$$

Then, by syntactical rewriting, the goal cannot be reduced because, ϵ_0 cannot be obtained from $\delta * (\epsilon + |z| + 1) + |x| * \epsilon$ by a substitution. However, in this situation, the goal can be reduced to

$$\underset{\delta,\epsilon}{\exists} (\delta * (\epsilon + |g_0[x_0]| + 1) + |f_0[x_0]| * \epsilon = \epsilon_0 \wedge |f_0[y] - f_0[x_0]| < \delta \wedge |g_0[y] - g_0[x_0]| < \epsilon).$$

It turns out that the PCS method is quite powerful for generating, with almost no superfluous search, natural (easy to

understand) proofs for many elementary "alternating quantifiers theorems". (Note, however, that these "elementary" theorems give lots of headache to beginning students of mathematics. Also, they are outside the scope of both purely algebraic provers like Collins' cad-method and the usual general predicate logic provers. Thus, being able to generate natural proofs for these theorems, in our view, is a definite step forward.)

We demonstrate the method by showing the proof generated by the PCS prover of TH∃OREM∀ for the proposition introduced in Section 2.

In fact, the text actually produced by the TH∃OREM∀ PCS prover (in a *Mathematica* notebook) is structured much more nicely than what we can show here in the narrow columns of this paper. Also, the actual notebook generated displays "nested cell brackets" that can be used to browse the proof in a "structured way", as was explained in [8].

Note that the entire proof including all intermediate natural language text (i.e. everything between the two horizontal lines below) is generated completely automatically by the PCS prover:

Prove:

$$\text{(cont*)}\quad \underset{f,g,x}{\forall}\ (\text{continuous}[f,x] \wedge \text{continuous}[g,x] \Rightarrow \text{continuous}[f*g,x])$$

under the assumptions:

$$\text{(c2:)}\quad \underset{f,x}{\forall}\left(\text{continuous}[f,x] \Leftrightarrow \underset{\epsilon>0}{\forall}\ \underset{\delta>0}{\exists}\ \underset{|y-x|<\delta}{\underset{y}{\forall}}\ (|f[y]-f[x]|<\epsilon)\right),$$

$$\text{(f*g)}\quad \underset{f,g,x}{\forall}\ ((f*g)[x] = f[x]*g[x]),$$

$$\text{(dist*)}\quad \underset{x,y,z,t,\delta,\epsilon}{\forall}\ (|y*t-x*z|<\delta*(\epsilon+|z|+1)+|x|*\epsilon \Leftarrow (|y-x|<\delta \wedge |t-z|<\epsilon)),$$

$$\text{(min>)}\quad \underset{m,M1,M2}{\forall}\ (\min[M1,M2]>m \Leftrightarrow M1>m \wedge M2>m),$$

$$\text{(<min)}\quad \underset{m,M1,M2}{\forall}\ (m<M1 \wedge m<M2 \Leftrightarrow m<\min[M1,M2]).$$

We assume

(1) $\text{continuous}[f_0,x_0] \wedge \text{continuous}[g_0,x_0]$,

and show

(2) $\text{continuous}[f_0*g_0,x_0]$.

Formula (1.1), by (c2:), implies

$$(3)\quad \underset{\epsilon>0}{\forall}\ \underset{\delta>0}{\exists}\ \underset{|y-x_0|<\delta}{\underset{y}{\forall}}\ (|f_0[y]-f_0[x_0]|<\epsilon).$$

By (3) we can introduce a Skolem function such that

$$(4)\quad \underset{\epsilon>0}{\forall}\left(\delta_0[\epsilon]>0\right) \wedge \underset{\epsilon>0}{\forall}\left(\underset{|y-x_0|<\delta_0[\epsilon]}{\underset{y}{\forall}}\ (|f_0[y]-f_0[x_0]|<\epsilon)\right).$$

Formula (1.2), by (c2:), implies

$$(5)\quad \underset{\epsilon>0}{\forall}\ \underset{\delta>0}{\exists}\ \underset{|y-x_0|<\delta}{\underset{y}{\forall}}\ (|g_0[y]-g_0[x_0]|<\epsilon).$$

By (5) we can introduce a Skolem function such that

$$(6)\quad \underset{\epsilon>0}{\forall}\left(\delta_1[\epsilon]>0\right) \wedge \underset{\epsilon>0}{\forall}\left(\underset{|y-x_0|<\delta_1[\epsilon]}{\underset{y}{\forall}}\ (|g_0[y]-g_0[x_0]|<\epsilon)\right).$$

Formula (2), using (c2:), is implied by

$$(9)\quad \underset{\epsilon>0}{\forall}\ \underset{\delta>0}{\exists}\ \underset{|y-x_0|<\delta}{\underset{y}{\forall}}\ (|(f_0*g_0)[y]-(f_0*g_0)[x_0]|<\epsilon).$$

We assume

(10) $\epsilon_0>0$,

and show

$$(11)\quad \underset{\delta>0}{\exists}\ \underset{|y-x_0|<\delta}{\underset{y}{\forall}}\ (|(f_0*g_0)[y]-(f_0*g_0)[x_0]|<\epsilon_0).$$

We have to find $\delta_2{}^*$ such that

$$(12)\quad \delta_2{}^*>0 \wedge \underset{|y-x_0|<\delta_2{}^*}{\underset{y}{\forall}}\ (|(f_0*g_0)[y]-(f_0*g_0)[x_0]|<\epsilon_0).$$

Formula (12), using (f*g), is implied by

$$(13)\quad \delta_2{}^*>0 \wedge \underset{|y-x_0|<\delta_2{}^*}{\underset{y}{\forall}}\ (|f_0[y]*g_0[y]-f_0[x_0]*g_0[x_0]|<\epsilon_0).$$

Formula (13), using (dist*), is implied by

$$(14)\quad \underset{\delta*(\epsilon+|g_0[x_0]|+1)+|f_0[x_0]|*\epsilon=\epsilon_0}{\underset{\delta,\epsilon}{\exists}}\left(\delta_2{}^*>0 \wedge \underset{|y-x_0|<\delta_2{}^*}{\underset{y}{\forall}}\ (|f_0[y]-f_0[x_0]|<\delta \wedge |g_0[y]-g_0[x_0]|<\epsilon)\right).$$

We have to find $\delta_2{}^*$, $\delta_3{}^*$, $\epsilon_1{}^*$ such that

$$(15)\quad (\delta_3{}^**(\epsilon_1{}^*+|g_0[x_0]|+1)+|f_0[x_0]|*\epsilon_1{}^*=\epsilon_0) \wedge \delta_2{}^*>0 \wedge \underset{|y-x_0|<\delta_2{}^*}{\underset{y}{\forall}}\ (|f_0[y]-f_0[x_0]|<\delta_3{}^* \wedge |g_0[y]-g_0[x_0]|<\epsilon_1{}^*).$$

Formula (15), using (4.2) and (6.2), is implied by

$$(\delta_3{}^**(\epsilon_1{}^*+|g_0[x_0]|+1)+|f_0[x_0]|*\epsilon_1{}^*=\epsilon_0) \wedge \delta_2{}^*>0 \wedge \underset{|y-x_0|<\delta_2{}^*}{\underset{y}{\forall}}\ ((\delta_3{}^*>0 \wedge |y-x_0|<\delta_0[\delta_3{}^*]) \wedge (\epsilon_1{}^*>0 \wedge |y-x_0|<\delta_1[\epsilon_1{}^*])),$$

which, using (<min), is implied by

$$(16)\quad (\delta_3{}^**(\epsilon_1{}^*+|g_0[x_0]|+1)+|f_0[x_0]|*\epsilon_1{}^*=\epsilon_0) \wedge \delta_2{}^*>0 \wedge \underset{|y-x_0|<\delta_2{}^*}{\underset{y}{\forall}}\ (\delta_3{}^*>0 \wedge \epsilon_1{}^*>0 \wedge |y-x_0|<\min[\delta_0[\delta_3{}^*],\delta_1[\epsilon_1{}^*]]).$$

Formula (16) is implied by

$$(17)\quad \begin{array}{l}(\delta_3{}^* * (\epsilon_1{}^* + |g_0[x_0]| + 1) + |f_0[x_0]| * \epsilon_1{}^* = \epsilon_0) \wedge \\ \delta_2{}^* > 0 \wedge \delta_3{}^* > 0 \wedge \epsilon_1{}^* > 0 \wedge \\ \quad \underset{\substack{y \\ |y-x_0|<\delta_2{}^*}}{\forall} \quad (|y - x_0| < \min[\delta_0[\delta_3{}^*], \delta_1[\epsilon_1{}^*]]).\end{array}$$

Partially solving it, formula (17) is implied by

$$(18)\quad \begin{array}{l}(\delta_3{}^* * (\epsilon_1{}^* + |g_0[x_0]| + 1) + |f_0[x_0]| * \epsilon_1{}^* = \epsilon_0) \wedge \\ \min[\delta_0[\delta_3{}^*], \delta_1[\epsilon_1{}^*]] > 0 \ \wedge \ \delta_3{}^* > 0 \ \wedge \ \epsilon_1{}^* > 0 \wedge \\ (\delta_2{}^* = \min[\delta_0[\delta_3{}^*], \delta_1[\epsilon_1{}^*]]).\end{array}$$

Formula (18), using (min>), is implied by

$$\begin{array}{l}(\delta_3{}^* * (\epsilon_1{}^* + |g_0[x_0]| + 1) + |f_0[x_0]| * \epsilon_1{}^* = \epsilon_0) \wedge \\ (\delta_0[\delta_3{}^*] > 0 \wedge \ \delta_1[\epsilon_1{}^*] > 0) \wedge \delta_3{}^* > 0 \ \wedge \ \epsilon_1{}^* > 0 \wedge \\ (\delta_2{}^* = \min[\delta_0[\delta_3{}^*], \delta_1[\epsilon_1{}^*]]),\end{array}$$

which, using (4.1) and (6.1), is implied by

$$(19)\quad \begin{array}{l}(\delta_3{}^* * (\epsilon_1{}^* + |g_0[x_0]| + 1) + |f_0[x_0]| * \epsilon_1{}^* = \epsilon_0) \wedge \\ \delta_3{}^* > 0 \wedge \epsilon_1{}^* > 0 \wedge (\delta_2{}^* = \min[\delta_0[\delta_3{}^*], \delta_1[\epsilon_1{}^*]]).\end{array}$$

Summarizing, we reduced the proof to a solving problem. We have to find $\delta_2{}^*$, $\delta_3{}^*$, $\epsilon_1{}^*$ such that (19) holds under the current knowledge. The problem can be solved by calling the *Mathematica* implementation of the Cylindrical Algebraic Decomposition Algorithm. Hence, we are done. The solution is of the form

$0 < \ \delta_3{}^* < \frac{\epsilon_0}{1+|g_0[x_0]|}$,

$\epsilon_1{}^* = \frac{\epsilon_0 - (1+|g_0[x_0]|) * \delta_3{}^*}{\delta_3{}^* + |f_0[x_0]|}$,

$\delta_2{}^* = \min[\delta_0[\delta_3{}^*], \delta_1[\epsilon_1{}^*]]$.

In fact, the above proof generated by the TH∃OREM∀ PCS prover produces much more detailed information on the "solving terms" than is normally done in proofs by human mathematicians. This information is quite interesting and would make it possible to formulate a much stronger version of the proposition, namely a version in which the dependence of the objects to be found (like $\epsilon_1^*, \delta_1^*, \delta_3^*$) on the objects given (like ϵ_0) is explicit. Also, the above proof follows the style of proving, in which, during the generation of the proof, full motivation is given why certain constructions are done in the way displayed. This style of presenting proofs is often considered to be "more pedagogical" than the style where the constructions are presented "by the teacher" without any motivation and the student afterwards is left (with the easy) task of just verifying that the constructions are appropriate.

However, as soon as this detailed and explicit version of the proof is generated it would be possible (by a "proof simplification" algorithm) to translate the proof into a version which suppresses the details of the construction and/or rearranges the proof into a style where the constructions are just given and then verified.

5 The TH∃OREM∀ Set Theory Prover

The PCS approach to proving is not limited to general predicate logic provers. It can easily be extended to the design of special provers, for example a prover for set theory. For this, we insert special rules for the functions and predicates of set theory into the prove phase of the PCS cycle:

- $x \in \{T_y \mid_y P_y\}$ can be reduced to $\underset{y}{\exists} P_y \ \wedge \ x = T_y$
- $A \subseteq B$ can be reduced to $\underset{x}{\forall} x \in A \Rightarrow x \in B$
- $A = B$ can be reduced to $A \subseteq B \ \wedge \ B \subseteq A$

On the other hand,

$(\underset{A}{\exists} x \in A \ \wedge \ A \subseteq B) \Rightarrow x \in B$

can be used as a rule in the compute phase, i.e. it can be used to rewrite $x \in X \ \wedge \ X \subseteq Y$ in the knowledge base into $x \in Y$ or to "reduce" the goal $x \in X$ to the goal $\underset{A}{\exists} x \in A \ \wedge \ A \subseteq X$. Practically, the rule reduces the goal $x \in X$ to $x \in A$ in case we have $A \subseteq X$ in the knowledge base and to $A \subseteq X$ in case we have $x \in A$ in the knowledge base.

We will present an example of a simple proof generated by the TH∃OREM∀ set theory prover, which, for obvious reasons, is not an isolated prover but combines with the predicate logic prover: We start out with an arbitrary but fixed binary relation $\sim$.

Axiom["Transitivity", any$[r, s, t]$,
$r \sim s \wedge s \sim t \Rightarrow r \sim t$ "trans"]

Definition["Equivalence class", any$[a]$,
$\lfloor a \rfloor := \{b \mid_b b \sim a\}$ "class"].

We now show, for example, the following lemma:

Lemma["Classes inclusion", any$[a, b]$,
$a \sim b \Rightarrow \lfloor a \rfloor \subseteq \lfloor b \rfloor$ "incl"]

`Prove`[Lemma["Classes inclusion"], using→
⟨Axiom["Transitivity"], Definition["Equivalence class"]⟩,
by→SetTheoryPCProver]

Note that the entire proof including all intermediate natural language text (i.e. everything between the two horizontal lines below) is generated completely automatically by "SetTheoryPCProver":

Prove:

(incl) $\underset{a,b}{\forall} (a \sim b \Rightarrow \lfloor a \rfloor \subseteq \lfloor b \rfloor)$,

under the assumptions:

(trans) $\underset{r,s,t}{\forall} (r \sim s \wedge s \sim t \Rightarrow r \sim t)$,

(class) $\lfloor a \rfloor := \{b \mid_b b \sim a\}$.

We assume

(1) $a_0 \sim b_0$

and show

(2) $\lfloor a_0 \rfloor \subseteq \lfloor b_0 \rfloor$.

For proving (2) we assume

(3) $a3_0 \in \lfloor a_0 \rfloor$,

and show

(4) $a3_0 \in \lfloor b_0 \rfloor$.

Formula (3), by (class), implies

(5) $a3_0 \in \{b \mid_b b \sim a_0\}$.

From (5) we obtain

(6) $a3_0 \sim a_0$.

Formula (4), using (class), is implied by

(7) $a3_0 \in \{b \mid_b b \sim b_0\}$.

In order to prove (7) we have to show

(8) $a3_0 \sim b_0$.

Because the conclusion of (trans) matches the goal (8), we specialize (trans) to

(9) $\underset{s}{\forall}\,(a3_0 \sim s \wedge s \sim b_0 \Rightarrow a3_0 \sim b_0)$.

By (9), for proving (8) it suffices to prove

(10) $\underset{s}{\exists}\,(a3_0 \sim s \wedge s \sim b_0)$.

For proving (10), it suffices to prove

(11) $a3_0 \sim a_0 \wedge a_0 \sim b_0$.

Proof of (11.1): This formula is identical to (6).

Proof of (11.2): This formula is identical to (1).

---

6 Special Provers Within TH∃OREM∀ : Groebner Bases Prover and Gosper-Zeilberger Prover

Special provers (solvers, and simplifiers) can be be integrated into TH∃OREM∀. A special prover P for a theory (i.e. a collection of formulae) T is a prover satisfying the "Correctness Meta-Theorem for P w.r.t. T":

> *For all knowledge bases K and goals G, if P produces a proof of G under the assumption K, then G is a logical consequence of $K \bigcup T$.*

In fact, some of the algebraic provers we are currently interested in, like the Groebner bases prover and Collins' prover, are also complete.

Similarly, the correctness of special solvers and simplifiers for a given theory T is defined.

Typically, the reduction of a proof problem to a strong special prover (or solver) is a process that deserves explanation in a proof text whereas the actual call of the special prover (e.g. the computation of a Groebner basis) is not something the user is interested to see in detail. Rather, one normally prefers to use these provers in a "black box" style. We give two examples: The automated proof of universally quantified boolean combinations of equalities over the complex numbers using the Groebner bases method and the automated proof of combinatorial identities using the Gosper-Zeilberger prover with the extension of this method by Paule and Schorn, see [17]. Both methods are now accessible by calls from within TH∃OREM∀, i.e. the proof problems can be formulated in the TH∃OREM∀ formal language, the reduction of the proof problems to the black box methods is explained by natural language text in the proofs generated and the actual black box computation are then just presented by their result.

We first show the formal text that produces a proof by the Groebner bases method:

Formula["Boolean",

$$\underset{x,y}{\forall}\Big((x^2*y-3*x\neq 0)\vee(x*y+x+y\neq 0)\vee \big((x^2*y+3*x=0)\vee((-2)*x^2+(-7)*x*y+ x^2*y+x^3*y+(-2)*y^2+(-2)*x*y^2+ 2*x^2*y^2=0)\big)\wedge\big(x^2+(-x)*y+ x^2*y+(-2)*y^2+(-2)*x*y^2=0\big)\Big)$$

"1"],

`Prove`[Formula["Boolean"], by→ GroebnerBasisProver].

Note that, again, the entire proof including all intermediate natural language text (i.e. everything between the two horizontal line below) is generated completely automatically by the prover.

Prove:

$$\text{(1)}\quad \underset{x,y}{\forall}\,\big(x^2*y-3*x\neq 0\big)\vee(x*y+x+y\neq 0)\vee \big(\big(x^2*y+3*x=0\big)\vee \big((-2)*x^2+(-7)*x*y+x^2*y+ x^3*y+(-2)*y^2+(-2)*x*y^2+ 2*x^2*y^2=0\big)\big)\wedge \big(x^2+(-x)*y+x^2*y+(-2)*y^2+ (-2)*x*y^2=0\big)$$

with no assumptions.

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:

$$\text{(1.1)}\quad \underset{x,y}{\forall}\,\big(\big(x^2+(-x*y)+x^2*y+-2*y^2+ -2*x*y^2=0\big)\vee\big(-3*x+x^2*y\neq 0\big)\vee (x+y+x*y\neq 0)\big),$$

$$\text{(1.2)}\quad \underset{x,y}{\forall}\,\big((3*x+x^2*y=0)\vee \big(-2*x^2+-7*x*y+x^2*y+x^3*y+ -2*y^2+-2*x*y^2+2*x^2*y^2=0\big)\vee \big(-3*x+x^2*y\neq 0\big)\vee(x+y+x*y\neq 0)\big).$$

We now prove the above individual problems separately:

Proof of (1.1):

Here, in the output, the proof of (1.1) is generated. In this paper, for reasons of space, we do not show this proof. Rather, we only show the proof of (1.2), which is more interesting because it introduces two slack variables.

Proof of (1.2):

This proof problem has the following structure:

(1.2.structure)

$$\underset{x,y}{\forall}\,((\text{Poly1}[1]\neq 0)\vee (\text{Poly1}[2]\neq 0)\vee(\text{Poly1}[3]=0)\vee(\text{Poly1}[4]=0)),$$

where

$$\begin{aligned}\text{Poly1[1]} &= -3*x+x^2*y\\ \text{Poly1[2]} &= x+y+x*y\\ \text{Poly1[3]} &= 3*x+x^2*y\\ \text{Poly1[4]} &= -2*x^2+-7*x*y+x^2*y+x^3*y+\\ &\quad -2*y^2+-2*x*y^2+2*x^2*y^2.\end{aligned}$$

(1.2.structure) is equivalent to

(1.2.implication)
$$\underset{x,y}{\forall}\ ((\text{Poly1[1]}=0)\wedge(\text{Poly1[2]}=0)\Rightarrow (\text{Poly1[3]}=0)\vee(\text{Poly1[4]}=0)).$$

(1.2.implication) is equivalent to

(1.2.not-exists)
$$\neg\ \underset{x,y}{\exists}\ (((\text{Poly1[1]}=0)\wedge(\text{Poly1[2]}=0))\wedge((\text{Poly1[3]}\neq 0)\wedge(\text{Poly1[4]}\neq 0))).$$

By introducing the slack variable(s) $\{\xi 4, \xi 5\}$ (1.2.not-exists) is transformed into the equivalent formula

(1.2.not-exists-slack)
$$\neg\ \underset{\xi 4,\xi 5}{\exists}\ (((\text{Poly1[1]}=0)\wedge(\text{Poly1[2]}=0))\wedge\{-1+\xi 4\ \ \text{Poly1[3]}=0, -1+\xi 5\ \ \text{Poly1[4]}=0\}).$$

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

$$\{\text{Poly1[1]}, \text{Poly1[2]}, -1+\xi 4\,\text{Poly1[3]}, -1+\xi 5\,\text{Poly1[4]}\}$$

is exactly $\{1\}$.

The Groebner basis is :
$\{1\}$.

Hence, (1.2) is proved.

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

We now show the formal text that produces a proof by the Gosper-Zeilberger method. Note that, actually, this method is more than a prover, it is a theorem generator (or, in other words, a simplifier): For a given summation term, it produces the "closed form", i.e. a simplified term or recurrence. The example we give, shows the power of the method: It produces a closed form for a sum whose evaluation was proposed as a SIAM REVIEW problem, see [16].

Formula["SIAM series",
$\sum_{k=1}^{n} \frac{(-1)^{k+1}(4k+1)(2k)!}{2^k(2k-1)(k+1)!2^k k!}$]

`Prove`[Formula["SIAM series"],

by→GosperZeilbergerProver].

Note that, again, the entire proof including all intermediate natural language text (i.e. everything between the two horizontal line below) is generated completely automatically by the prover.

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

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

Proof:
Let Δ_k denote the forward difference operator in k. Then the Theorem follows from summing the equation

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

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:

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

to 1.

---

7 Extending Existing Provers by Meta-Strategies

Given a prover P, one can apply various strategies for enhancing the proving power of P. One of these strategies is what we call the "cascade": Intuitively, the idea is that, given a goal G and a knowledge base K, we let P try to find a proof. If P succeeds, we stop and present the proof. If not, we let a "failure analyzer" analyze the proof attempt and conjecture a lemma L, which could be strong enough to allow P to prove G from $K \bigcup L$. Now we let P try to prove L from K.

If it succeeds we let P try, again, to prove G but this time under the assumption $K \bigcup L$.

If it fails we let the failure analyzer work on the failing proof.

More formally, given a prover P and a "conjecture from failure generator" C, the following recursive "cascade" may result in a much stronger prover that, in fact, does not only prove more theorems than P but, on the way of proving a goal from a knowledge base, gradually extends the knowledge base by "useful" lemmas:

```
Cascade[G,K,C,P]:=
    proof-attempt=Prove[G,K,P];
    if proof-attempt is successful,
            then Return[{"proved",K}];
            else L=C[proof-attempt];
    {proof-value,new-K}=Cascade[L,K,C,P];
    if proof-value="proved",
            then Cascade[G,K ⋃ L,C,P];
            else
                    if new-K=K,
                            then Return[{"failed",K}];
                            else Cascade[G,new-K,C,P].
```

We show the effect of the cascade in the case of a simple induction prover for natural numbers and a simple conjecture generator that conjectures a generalized equality over the natural numbers from a special instance of the equality, which occurs in a failing proof.

Starting from the definition

Definition["Addition",
$\underset{m}{\forall}\ \ m+0=0$
$\underset{m,n}{\forall}\ \ m+n^+=(m+n)^+$]

we might want to prove

Proposition["Commutativity +",

$\forall_{m,n}\ m+n=n+m$].

This can be tried by calling

`Prove`[Proposition["Commutativity +"],
using→Definition["Addition"],
by→NNEqIndProver]].

With this simple prover, the proof will fail: It generates a proof attempt that is stuck at the situation where it should prove (for a constant m_1)

$$\text{(F)} \qquad (0+m_1)^+ = m_1^+$$

A human reader would immediately conjecture that, maybe, 0 is also a left unit, i.e.

$$\text{(L)} \qquad \forall_m\ 0+m=m$$

and that, maybe, if this conjecture is true, the proof of commutativity could go on and could be completed. Producing the conjecture (L) from the failure line is a relatively easy process: "Strip off all identical outer symbols from the left-hand and right-side terms of (F) and turn the constants into variables". With this simple procedure, which we programmed and called ConjectureGenerator, the call of the above cascade

`Prove`[Proposition["Commutativity +"],
using→Definition["Addition"],
by→Cascade[NNEqIndProver,ConjectureGenerator]]

produces, successively and without any further user-interaction, five notebooks that contain the following proofs and proof attempts:

1) A failing proof attempt for proving the goal

   $\forall_{m,n}\ m+n=n+m$

   from the knowledge base

   $\forall_m\ m+0=m,$
   $\forall_{m,n}\ m+n^+=(m+n)^+.$

2) A successful proof for the goal

   $\forall_m\ 0+m=m$

   (which is automatically generated from analyzing the failing proof 1)) from the knowledge base

   $\forall_m\ m+0=m,$
   $\forall_{m,n}\ m+n^+=(m+n)^+.$

3) A failing proof attempt for proving the goal

   $\forall_{m,n}\ m+n=n+m$

   from the knowledge base

   $\forall_m\ 0+m=m,$
   $\forall_m\ m+0=m,$
   $\forall_{m,n}\ m+n^+=(m+n)^+.$

4) A successful proof for the goal

   $\forall_{n,m}\ n^+ + m = (n+m)^+$

   (which is automatically generated from analyzing the failing proof 3)) from the knowledge base

   $\forall_m\ 0+m=m,$
   $\forall_m\ m+0=m,$
   $\forall_{m,n}\ m+n^+=(m+n)^+.$

5) A successful proof for the goal

   $\forall_{m,n}\ m+n=n+m$

   from the knowledge base

   $\forall_{n,m}\ n^+ + m = (n+m)^+,$
   $\forall_m\ 0+m=m,$
   $\forall_m\ m+0=m,$
   $\forall_{m,n}\ m+n^+=(m+n)^+.$

Note that, in addition to being able to prove the original goal, the combination of NNEqIndProver with ConjectureGenerator in the cascade also automatically produces two additional, and in fact quite natural and interesting, lemmata.

8 Proof Simplification

As a natural strategy, when human provers are confronted with a proof problem, one tries to find a first version of a proof, which might be unnecessarily complicated. Then, one works on the proof found and tries to simplify it in various ways, for example by extracting similar proof parts from various parallel branches of the proof or by eliminating subgoals that did not actually contribute to the ultimate proof goal. We are implementing such proof simplification strategies that work, as post-processors, on the proof objects generated by our provers, in particular the predicate logic prover.

For lack of space we do not present in this paper examples of proofs and of their simplified versions (many examples are available at `www.theorema.org`), however we explain below the techniques we are using.

Removing superfluous branches. Certain deduction steps generate alternative branches of proofs. For instance, when the goal is the right-hand side of an assumed implication, one proof alternative is to prove the left-hand side (which is natural, but may not succeed), and another alternative is to assume the negation of the left-hand side (which is less natural). By simplification of a proof, only the branches which are effectively useful for the success are shown.

Removing superfluous steps. During the search for the proof some formulae may be generated which finally are not necessary for proving the goal. By simplification the deduction steps producing these formulae are removed as described below. During the generation of the proof object, auxiliary information is stored showing, for each deduction step, which existing formulae are used for producing which new formulae. The proof–simplification routine starts from the final proof steps and collects backwards those formulae

which have been actually used, while marking the corresponding proof steps. After that, the unmarked proof steps are removed.

The following simplification techniques are currently under implementation.

Removing duplicate steps. Certain deduction steps split the proof in several necessary branches (e. g. proving a conjunction is done by proving each conjunct separately). It may happen that on several branches the same deduction steps are repeated. This proof-simplification technique moves these common deduction steps above the splitting point in the proof object.

Combining often–used proof steps. For instance, if the goal is an universally quantified implication, then one proof step transforms the goal by eliminating the universal quantifier (the corresponding variables are replaced by new constants), and the next proof step adds the left-hand side of the implication to the assumptions and replaces the goal by the right-hand side. The proof looks more natural if the two steps are combined into one – and this is currently done by having one special rule for such goals. However, since there are many combinations of rules which should be treated in this way, introducing new rules for each combination may increase too much the number of rules of the prover, with negative impact on complexity of the code and on the performance. Therefore it is preferable to perform the combination of successive steps in the post-processing phase, when the proof is already found and only the successful branches need to be examined.

Didactical re-arrangement of proofs: It is well known that, from a pedagogical point of view, one can distinguish two proof styles:

- The *follow-the-invention style*: The proof problem is gradually transformed in such a way that the non-trivial constructions necessary during the proof are logically well motivated. After the constructions are then given (by invoking a "solver") verification of the construction is not any more necessary.
- The *present-and-verify style*: At certain stages in the proof, "suitable constructions" must be offered and, afterwards, verified by sub-proofs.

One can give pedagocially relevant arguments in favor of both proof styles. The PCS provers, which we proposed in this paper, produce proofs in the "follow-the-invention" style (see the example in Section 4). We will construct a translating algorithm that translate such proofs into the "present-and-verify" style so that the user has the option of chosing in which style he wants to see and study the proofs.

9 Conclusion

We reported on some of the new features of TH∃OREM∀. The next steps in the TH∃OREM∀ project will mainly concetrate on improving the PCS prover(s) and implementing and integrating various special provers described in the literature that have proven to be particularly successful. TH∃OREM∀, in its present version (Version 1), offers a fixed arsenal of provers (solvers, and simplifiers). We are also working on a major re-design of TH∃OREM∀ (Version 2) that will make it possible for the user to formulate his own provers (solvers, and simplifiers) in a language that will be particularly suited for this task, i.e. will make it particularly easy to program provers (sovers, and simplifiers) that, in addition to the abstract proof objects, also produce natural language intermediate explanatory proof texts.

References

[1] Armando, A., Coglio, A., and Giunchiglia, F. The Control Component of Open Mechanized Reasoning Systems. In *Proceedings of CALCULEMUS: Systems for Integrated Computation and Deduction* (Trento, Italy, 1999), A. Armando and T. Jebelean, Eds., Electronics Notes in Theoretical Computer Science, Elsevier. See also http://www.mrg.dist.unige.it/omrs/.

[2] Bauer, A., Clarke, E. M., and Zhao, X. Analytica: - An Experiment in Combining Theorem Proving and Symbolic Computation. In *International Conference on Artificial Intelligence and Symbolic Mathematical Computation, AISMC-3, Steyr, Austria* (1996), pp. 21–37.

[3] Benzmueller, C., Cheikhrouhou, L., Fehrer, D., Fiedler, A., Huang, X., Kerber, M., Kohlhase, M., Konrad, K., Mellis, E., Meier, A., Schaarschmidt, W., Siekman, J., and Sorge, V. ΩMEGA: Towards a Mathematical Assistant. In *Proceedings of the 14th Conference on Automated Deduction* (Townsville, Australia, 1997), W. McCune, Ed., no. 1249 in LNAI, Springer-Verlag, pp. 252–255. See also http://www.ags.uni-sb.de/projects/deduktion/.

[4] Bertoli, P. G., Calmet, J., Giunchiglia, F., and Homann, K. Specification and Integration of Theorem Provers and Computer Algebra Systems. *Fundamenta Informaticae* (1999). Accepted for publication.

[5] Boyer, R., and Moore, J. *A Computational Logic.* Academic Press, 1979. See also http://www.cli.com/software/nqthm/.

[6] Buchberger, B. The Objectives of the *Theorema* Project. Talk at the 1996 CALCULEMUS Meeting, University of Rome, Italy, 1996.

[7] Buchberger, B. Theory Exploration versus Theorem Proving. In *Proceedings of CALCULEMUS: Systems for Integrated Computation and Deduction* (Trento, Italy, 1999), A. Armando and T. Jebelean, Eds., Electronics Notes in Theoretical Computer Science, Elsevier.

[8] Buchberger, B., Jebelean, T., Kriftner, F., Marin, M., Tomuţa, E., and Văsaru, D. A Survey of the *Theorema* Project. In *Proceedings of ISSAC'97 (International Symposium on Symbolic and Algebraic Computation)* (Maui, Hawaii, 1997), W. Kuechlin, Ed., ACM Press, pp. 384–391.

[9] Bundy, A., van Harmelen, F., Horn, C., and Smaill, A. The Oyster-Clam system. In *Proceedings of the 10th International Conference on Automated Deduction* (1990), M. E. Stickel, Ed., Springer-Verlag, pp. 647–648.

[10] Caprotti, O., and Cohen, A. M. Integrating Computational and Deduction Systems Using *OpenMath*. In *Proceedings of CALCULEMUS: Systems for Integrated Computation and Deduction* (Trento, Italy, 1999), A. Armando and T. Jebelean, Eds., Electronics Notes in Theoretical Computer Science, Elsevier. See also http://www.nag.co.uk/projects/openmath/omsoc/.

[11] Collins, G. E. Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition. In *Second GI Conference on Authomata Theory and Formal Languages* (1975), vol. 33 of *LNCS*, Springer-Verlag, Berlin, pp. 134–183.

[12] Constable, R. L., et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986. See also http://www.cs.cornell.edu/Info/Projects/NuPrl.

[13] Gordon, M., and T.F.Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993. See also http://www.cl.cam.ac.uk/Research/HVG/HOL.

[14] Huet, G., Kahn, G., and Paulin-Mohring, C. *The Coq Proof Assistant. A Tutorial. Version 5.10*. INRIA-Rocquencourt, CNRS-ENS Lyon, 1994. See also http://coq.inria.fr/.

[15] Mellis, E. The "Limit" Domain. In *Proceedings of AIPS-98* (1998), R. Simmons, M. Veloso, and S. Smith, Eds.

[16] Paule, P. Problem 94-2. *SIAM REVIEW* (1995), 105–106.

[17] Paule, P., and Schorn, M. A Mathematica Version of Zeilberger's Algorithm for Proving Binomial Coefficient Identities. *J. Symbolic Computation 20* (1995), 673–698.

[18] Paulson, L. C. *Introduction to Isabelle*. Computer Laboratory, University of Cambridge, 1996. See also http://www.cl.cam.ac.uk/Research/HVG/Isabelle/.

[19] Richardson, J. D. C., Smaill, A., and Green, I. M. System Description: Proof Planning in Higher-order Logic with λ-Clam. In *Proceedings of CADE-15* (1998), C. Kirchner and H. Kirchner, Eds., vol. 1421 of *LNCS*, Springer Verlag, pp. 647–648.

[20] Strzebonski, A. Solving Equations and Inequalities with *Mathematica*. In *Proceedings of the* Mathematica *Developer Conference* (1999).

[21] Trybulec, A., and Blair, H. Computer Assisted Reasoning with MIZAR. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence* (Los Angeles, CA, Aug. 1985), A. Joshi, Ed., Morgan Kaufmann, pp. 26–28. See also http://mizar.uw.bialystok.pl/.

[22] Wolfram, S. *The Mathematica Book*. Wolfram Media and Cambridge University Press, 1999.