The TH∃OREM∀ System: Current Status and the Proving-Solving-Computing Cycle *

B. Buchberger, D. Văsaru, T. Jebelean

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

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

May 24, 2000

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.

We summarize the main features of the TH∃OREM∀ system: the formal text language, the computational sessions, propositional and predicate-logic proving, proving by induction and by simplification, the set theory prover, special provers (Groebner Bases, Zeilberger, external), the cascade-meta-strategy for the TH∃OREM∀ provers, proof simplification, and the Prove-Compute-Solve (PCS) prover. We discuss in more detail the PCS prover which is based on a combination of techniques for predicate-logic proving, rewriting using universal formulae, and solving in specific domains using black-box engines.

1 Introduction

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

*This research is partially supported by the INTAS project 76-960, the Austrian Science Foundation (FWF-SFB-project FO-1302), and the Upper Austrian Government (project "PROVE").

- 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.

The TH∃OREM∀ system is implemented using the computer algebra system *Mathematica* and it is based on earlier work of Buchberger [6, 8]. The first version of the system is described in [11] and other papers (see `www.theorema.org`), and the current version (in particular with more details about the features presented in this summary) is presented in [10].

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 efforts under way on this topic, notably from the research groups integrated in the EU Calculemus Consortium: [1], [3], [4], [12], [13], [22], [24]. Other projects with similar objectivess are described in [2], [5], [15], [16], [17], [21].

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 [9].

- 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* [25]) as our programming environment so that TH∃OREM∀ is available on all platforms.

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

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

- the formal text language for defining new notions, algorithms and their properties,
- the distinction between TH∃OREM∀ standard sessions and TH∃OREM∀ computational sessions,
- propositional and predicate-logic proving,
- proving by induction and by simplification over integers and lists,
- the set theory prover,
- tools for proof simplification,
- the integration of powerful special provers (the Groebner Bases prover and the Gosper-Zeilberger prover) as well as of other "external" predicate logic provers,
- tools for extending existing provers by general strategies: the cascade strategy,
- a new predicate logic prover that integrates proving, simplifying and solving steps in a systematic way: the PCS prover.

We discuss in more detail in this paper the design and the implementation of the Proving-Solving-Computing (PCS) prover, which is aimed at proving in high-order predicate logic, but also in proving over specific domains (e. g. integers, reals). This prover combines techniques for predicate-logic proving, rewriting using universal formulae, and solving in specific domains using black-box engines.

For additional details about the system and for proof examples the reader is referred to the home page of the project: `www.theorema.org`.

2 Main Features of the TH∃OREM∀ System

2.1 The TH∃OREM∀ Formal Text Language

The core language (expression language) of TH∃OREM∀ ([11]) is a version of higher order predicate logic. The "TH∃OREM∀ Formal Text Language" allows to to compose, in a hierarchical way, large mathematical knowledge bases from individual expressions. Here are some typical examples of formal text written in the TH∃OREM∀ formal text language:

Definition["continuity", any$[f, x]$,

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

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*"]

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

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.

TH∃OREM∀ **environments** allow to enter knowledge into the system in a style similar to how definitions, propositions etc. are given in mathematical textbooks. The ingredients of an environment are a keyword ("Definition", "Lemma", "Theory", etc. .), a label for later reference, one or more mathematical statement(s) (formulae), an enumeration of the (free) variables occurring in the statement, and, if necessary, conditions on the variables or relations among them. Environments are 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".

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 is avoided. However, 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 "+").

The TH∃OREM∀ Command Language: "Execution" in a TH∃OREM∀ standard session means *proving* or *computing* w. r. t. a certain knowledge and using a certain prover, solver, "computer". Some typical 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"]}]

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.

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`.

2.2 The Computational Session

When using the TH∃OREM∀ system, it may happen that 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 the "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 2.1, 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, one uses 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$[A, B]$]:

$$A \ominus B := \{x \underset{x \in A}{|} x \notin B\}$$

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.

2.3 Propositional and Predicate Logic Proving

All the provers of the TH∃OREM∀ system are based on rewrite rules which apply to *proof situations*. A proof situation consists of a collection of assumptions (assumed true) and a goal formula (which has to be proven) $\langle\{\text{assumptions}\}, \text{goal}\rangle$. Each rewrite rule tries to imitate a proof deduction step which is used in practice by the working mathematician. For instance, the practical rule "For proving $A \wedge B$ we prove A and we prove B" is represented as a rewrite rule which transforms the proof situation $\langle\{\text{assumptions}\}, A \wedge B\rangle$ into the two proof situations $\langle\{\text{assumptions}\}, A\rangle$ and $\langle\{\text{assumptions}\}, B\rangle$. Most of the rules will, in fact, produce only one proof situation, however some of the rules produce several proof situations related by *AND* (as above), and some rules will produce proof alternatives related by *OR*. Some rules are terminating - like "the goal is among the assumptions" and produce final proof steps. Therefore, the proof is represented internally as an *AND-OR* tree, called *proof-object*, whose nodes contain annotations documenting the proof steps (these annotations are used later to produce a human readable proof). Final proof steps are the leaves of the tree. The management of the proof-object is done by a control mechanism which also allows to combine of several provers – e. g. simplification and induction, see below.

The propositional prover available in the system is based on a set of rules which are similar to the ones used in sequent calculus, some examples are below:

$$\langle\{A, \text{not}A, \ldots\}, G\rangle \longrightarrow \text{Proved}$$

$$\langle\{A, A \Rightarrow B, \ldots\}, G\rangle \longrightarrow \langle\{A, B, \ldots\}, G\rangle$$

$$\langle\{\ldots\}, G_0 \vee G_1 \vee \ldots \vee G_n\rangle \longrightarrow \langle\{\text{not}G_1, \ldots, \text{not}G_n, \ldots\}, G_0\rangle$$

$$\langle\{A \vee B, \ldots\}, G\rangle \longrightarrow \begin{array}{l}\langle\{A, \ldots\}, G\rangle \\ AND \\ \langle\{B, \ldots\}, G\rangle\end{array}$$

$$\langle\{A \Rightarrow G, \ldots\}, G\rangle \longrightarrow \begin{array}{l}\langle\{\ldots\}, A\rangle \\ OR \\ \langle\{\mathrm{not}\, A, \ldots\}, G\rangle\end{array}$$

As a general strategy, the prover will try to move the negation symbol inside the formulae and to split the assumptions and the goal until a final proof situation is found.

The predicate logic prover uses the rules of the propositional prover intermixed in a convenient way with new rules for predicate logic. Some example of such rules are listed below:

$$\langle\{\ldots\}, \underset{x}{\forall}\, P[x]\rangle \longrightarrow \langle\{\ldots\}, P[x_0]\rangle$$

$$\langle\{\underset{x}{\exists}\, P[x], \ldots\}, G\rangle \longrightarrow \langle\{P[x_0], \ldots\}, G\rangle$$

$$\langle\{P[a] \Longrightarrow Q[a], \underset{x}{\forall}\, P[x], \ldots\}, G\rangle \longrightarrow \langle\{Q[a], \underset{x}{\forall}\, P[x], \ldots\}, G\rangle$$

Rules as above are "simplifying rules" because they simplify the proof situation. Such rules are used also in the PCS prover (described in the sequel) in the "proving" phase.

However, one also needs rules which "complicate" the proof situation either by adding new assumptions, either by creating several branches. An example of an applications of such rules is:

$$\langle\{\underset{x}{\forall}\, P[x] \Longrightarrow Q[x], P[a], \ldots\}, G\rangle \longrightarrow \langle\{Q[a], \underset{x}{\forall}\, P[x] \Longrightarrow Q[x], P[a], \ldots\}, G\rangle$$

which corresponds to *forward reasoning*. By *backward reasoning* one replaces the current goal by using an universal assumption, but in order to insure completness of the prover one usually has to follow several alternatives.

The usage of such rules has to be done following a certain proof search strategy in order to insure completness and efficiency of the prover. One such strategy is defined by the PCS prover (described later), which uses special universal assumptions (implications, equivalences, equalities) as rewrite rules.

In order to approach proof problems which also have other types of universal assumptions, we have implemented a *level saturation* strategy for predicate logic proving, which is applicable both in *forward* mode and in *backward* mode or combined. For instance, in forward mode, after all the "simplifying rules" have been exhausted, in one proof step all possible new assumptions are produced based on existing ground literals and universal assumptions, using a general forward inference rule. Then the simplifying rules are applied again, etc. This strategy produces proofs for proof problems which can be solved using ground assumptions.

In dual fashion, the backward mode consists in replacing the goal at each with cycle with all possible alternatives which are determined by the universal assumptions. Combining the two strategies produces efficiently proofs for problems having ground goals or simple existential goals.

However, this is not sufficient, for instance, for problems contaning formulae with alternating quantifiers. Currently we are implementing a strategy which uses the analysis of the relationship between the existential goals and the universal assumptions in order to make the appropriate instantiations which allow then the application of simpler rules. Preliminary experiments show that this method can produce proofs for a large class of problems, especially when it is combined with the use of metavariables – see [18].

2.4 Proving by Induction and by Simplification

Currently two induction provers are implemented in the system: induction over natural numbers and induction over lists. Each prover has a version for proving equalities which operates in cooperation with a prover for equalities by simplification.

For instance, the induction prover over natural numbers accepts a universal property $\underset{n}{\forall} P[n]$ as the goal of the proof situation and first tries to prove it by predicate logic. If the proof does not succeed, than induction is tried over the variable x, by checking the base case $P[0]$ first. If the base case succeeds then the prover creates the induction hypothesis $P[n_0]$ (where n_0 is a new constant) and and tries to prove $P[n_0 + 1]$. In fact, the method proceeds recursively for goals with several universal variables, because at each step one of the variables will be transformed into a constant.

The induction over lists works in a similar way, with the appropriate choice for the base case and the successor function.

The induction for equalities uses in addition a special "simplifier" prover which is able to simplify expressions by rewriting using a given set of equalities. Here, the strategy consists in using first the simplifier for the current goal, by transforming the right-hand side and the left-hand side using the equalities which are available among the assumptions (in the *knowledge base*). If the transformed expressions are identical then the proof succeeds, otherwise the the new equality is proven by induction.

Using induction the TH∃OREM∀ system is able to prove most of the elementary properties of the (aximatically defined) operations over the natural numbers (addition, multiplication, powering, summation) as well as the elementary properties of operations on lists (concatenation, reversing, number of elements). Additionally, the induction provers are a good candidate for the meta-strategy of proving by cascade – see in the sequel.

2.5 The Set Theory Prover

The PCS approach (presented later in this paper) 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:

- $x \in \{T_y \mid \underset{y}{} P_y\}$ 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.

The set theory prover is under development and already can produce natural style proofs for properties of the empty set, power set, equivalence classes, etc.

2.6 Special Provers Using External Engines

The TH∃OREM∀ proving, computing, and solving capabilities can be easily enhanced by using built-in features of the *Mathematica* system or by conecting with external programs. (For instance the PCS prover uses the quantifier elimination algorithm for finding witnesses in proofs involving real numbers.)

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. Currently the system is able to produce:

- automated proofs of universally quantified boolean combinations of equalities over the complex numbers using the Groebner bases method;
- automated proof of combinatorial identities using the Gosper-Zeilberger prover with the extension of this method by Paule and Schorn, see [20].

Both methods are 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.

Additionally, the system can be used to call external provers (e. g. by resolution), starting from proof problems which are expressed in the TH∃OREM∀ formal language. Interfaces to the following provers are currently implemented: Otter, EQP, Gandalf, Bliksem, and Mace.

2.7 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.

Currently this strategy is implemented for the induction prover for equalities over natural numbers and uses 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.

This prover is already able to produce quite natural proofs for the elementary theory of addition, multiplication, etc. by generating automatically the same lemmata which are presented in classical textbooks. For instance, when proving the commutativity of addition starting from the axioms:

Definition["Addition",

$$\begin{array}{ll} \underset{m}{\forall} & m + 0 = 0 \\ \underset{m,n}{\forall} & m + n^+ = (m + n)^+ \end{array} \quad]$$

the prover will generate (and prove):

- the left-unit lemma: $(0 + m)^+ = m^+$
- the left-successor lemma: $n^+ + m = (n + m)^+$

2.8 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.

The following techniques are used:

- removing alternative branches which do not contribute to the success of the proof;
- removing steps which produce formulae that are not necessary for proving the goal (this happens especially when using the "level saturation" strategy);
- removing duplicate steps which appear on parallel branches of *AND* nodes;
- combining often–used proof steps (as e. g. instantiation of universal goal followed by split of implicative goal);
- combining logically dependent steps (e. g. using an implicative assumption which has a conjunctive premise).

3 The 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 [7]. Meanwhile, this class of propositions is used by various people as a test set for their systems, see for example [19].

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 [11]), 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 [11]) 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

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

This situation essentially specifies a "find problem": We have to find x^*, y^*, such that $G[x^*, y^*, ...]$ 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 $x, y, ...$, special solvers are called. For example, if $x, y, ...$ are variables ranging over real numbers we call (a full or specialized) version of Collins' cad-method (see [14]) (which is implemented in *Mathematica* version 4.0, see [23]).

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

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

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

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

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 2.1.

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 [11].

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 (without user interaction):

Prove:

(cont*) $\forall_{f,g,x}$ (continuous$[f, x] \wedge$ continuous$[g, x] \Rightarrow$ continuous$[f * g, x]$)

under the assumptions:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

We assume

(10) $\epsilon_0>0$,

and show

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

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

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

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

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

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

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

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

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

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

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

which, using (<min), is implied by

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

Formula (16) is implied by

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

Partially solving it, formula (17) is implied by

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

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

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

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

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

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 re-arranges the proof into a style where the constructions are just given and then verified.

4 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. Mathematica: A System for Doing Mathematics by Computer? In *Advances in the Design of Symbolic Computation Systems*, A. Miola and M. Temperini, Eds. Springer New York - Vienna, 1993.

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

[8] Buchberger, B. Using Mathematica for Doing Simple Mathematical Proofs. In *Proceedings of the 4th Mathematica Users' Conference, Tokyo, November 2, 1996* (1996), Wolfram Media Publishing. To appear.

[9] 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.

[10] Buchberger, B., Dupré, C., Jebelean, T., Kriftner, F., Nakagawa, K., asaru, D. V., and Windsteiger, W. The Theorema Project: A Progress Report. In *Proceedings of Calculemus 2000* (St. Andrews, UK, 2000), M. Kohlhase and M. Kerber, Eds., Electronic Notes in Computer Science.

[11] Buchberger, B., Jebelean, T., Kriftner, F., Marin, M., Tomuţa, E., and Văsaru, D. An Overview of the *Theorema* Project. To appear in proceedings of ISSAC 1997, 1997.

[12] 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.

[13] 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/.

[14] 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.

[15] Constable, R. L., et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[16] Gordon, M., and T.F.Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[17] Huet, G., Kahn, G., and Paulin-Mohring, C. *The Coq Proof Assistant. A Tutorial. Version 5.10*. INRIA-Rocquencourt, CNRS-ENS Lyon, 1994.

[18] Konev, B., and Jebelean, T. Usnig Metavaribles in Theorema Provers. In *Proceedings of Calculemus 2000* (St. Andrews, UK, 2000), M. Kohlhase and M. Kerber, Eds., Electronic Notes in Computer Science.

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

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

[21] Paulson, L. C. *Introduction to Isabelle*. Computer Laboratory, University of Cambridge, 1996.

[22] 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.

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

[24] 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/.

[25] Wolfram, S. *The Mathematica Book*. Wolfram Media and Cambridge University Press, 1996.