Algorithm-Supported Mathematical Theory Exploration: A Personal View and Strategy

Bruno Buchberger*

Research Institute for Symbolic Computation,
Johannes Kepler University Linz,
A-4040 Linz, Austria
bruno.buchberger@jku.at

Abstract. We present a personal view and strategy for algorithm-supported mathematical theory exploration and draw some conclusions for the desirable functionality of future mathematical software systems. The main points of emphasis are: The use of schemes for bottom-up mathematical invention, the algorithmic generation of conjectures from failing proofs for top-down mathematical invention, and the possibility to program new reasoners within the logic on which the reasoners work ("meta-programming").

1 A View of Algorithm-Supported Mathematical Theory Exploration

Mathematical theories are collections of formulae in some formal logic language (e.g. predicate logic). Mathematical theory exploration proceeds by applying, under the guidance of a human user, various algorithmic reasoners for producing new formulae from given ones and aims at building up (large) mathematical knowledge bases in an efficient, reliable, well-structured, re-usable, and flexible way. Algorithm-supported mathematical theory exploration may also be seen as the logical kernel of the recent field of "Mathematical Knowledge Management" (MKM), see [10] and [5]. In the past few decades, an impressive variety of results has been obtained in the area of algorithm-supported reasoning both in terms of logical and mathematical power as wells as in terms of software systems, see for example, ALF [18], AUTOMATH [12], COQ [2], ELF [21], HOL [13], IMPS [1],

* Sponsored by FWF (Österreichischer Fonds zur Förderung der Wissenschaftlichen Forschung; Austrian Science Foundation), Project SFB 1302 ("Theorema") of the SFB 13 ("Scientific Computing"), and by RICAM (Radon Institute for Computational and Applied Mathematics, Austrian Academy of Science, Linz). The final version of this paper was written while the author was a visiting professor at Kyoto University, Graduate School of Informatics (Professor Masahiko Sato). I would like to thank my coworkers A. Craciun, L. Kovacs, Dr. T. Kutsia, and F. Piroi for discussions on the contents of this paper, literature work, and help in the preparation of this paper.

Isabelle [20], Lego [17], Mizar [3], Nuprl [11], Omega [4]. We also made an effort in this area, see the Theorema [9] system.

However, as a matter of fact, these reasoning systems are not yet widely used by the "working mathematicians" (i.e. those who do math research and/or math teaching). This is in distinct contrast to the current math (computer algebra/numerics) systems like Mathematica, Maple, etc. which, in the past couple of years, have finally found their way into the daily practice of mathematicians. In this paper, we want to specify a few features of future systems for algorithm-supported mathematical theory exploration which, in our view, are indispensable for making these systems attractive for the daily routine of working mathematicians. These features are:

- *Integration of the Functionality of Current Mathematical Systems*: Reasoning systems must retain the full power of current numerics and computer algebra systems including the possibility to program one's own algorithms in the system.
- *Attractive Syntax*: Reasoning systems must accept and produce mathematical knowledge in attractive syntax and, in fact, in flexible syntax that can be defined, within certain limitations, by the user.
- *Structured Mathematical Knowledge Bases*: Reasoning systems must provide tools for building up and using (large) mathematical knowledge libraries in a structured way in uniform context with the algorithm libraries and with the possibility of changing structures easily.
- *Reasoners for Invention and Verification*: Reasoning systems must provide reasoners both for inventing (proposing) and for verifying (proving, disproving) mathematical knowledge.
- *Learning from Failed Reasoning*: The results of algorithmic reasoners (in particular, algorithmic provers) must be post-processable. In particular, also the results of failing reasoning attempts must be accessible for further (algorithmic) analysis because failure is the main source of creativity for mathematical invention.
- *Meta-Programming*: The process of (algorithm-supported) mathematical theory exploration is nonlinear: While exploring mathematical theories using known algorithmic reasoners we may obtain ideas for new algorithmic reasoners and we may want to implement them in our system and use them in the next exploration round. Hence, reasoning systems must allow "meta-programming", i.e. the algorithmic reasoners must be programmed basically in the same logic language in which the formulae are expressed on which the reasoners work.

I think it is fair to say that, in spite of the big progress made in the past couple of years, none of the current reasoning systems fulfills all the above requirements. In fact, some of the requirements are quite challenging and will need a lot more of both fundamental research and software technology. It is not the goal of this paper, to compare the various current systems (see the above references) w.r.t. to these six requirements. Rather, we will first summarize how we tried to fulfil the first three requirements in our Theorema system (see the web site

http://www.theorema.org/ and the papers cited the) and then, in the main part of the paper, we will sketch a few ideas (which are partly implemented and partly being implemented in THEOREMA) that may contribute to the other three requirements.

2 Integration of the Functionality of Current Mathematical Systems

Let us start from the fact that predicate logic is not only rich enough to express any mathematical proposition but it includes also, as a sublanguage, a universal programming language and, in fact, a practical and elegant programming language. For example, in THEOREMA syntax, the following formula

$$\forall_F(\textit{is-Gröbner-basis}[F] \Leftrightarrow \forall_{f,g\in F}\ \textit{reduced}[\textit{S-polynomial}[f,g],F]=0) \qquad (1)$$

can be read as a proposition (which can be proved, by the inference rules of predicate logic, from a rich enough knowledge base K) but it can also be read as an algorithm which can be applied to concrete input polynomial sets F, like $\{x^2y-x-2, xy^2-xy+1\}$. Application to inputs proceeds by using a certain simple subset of the inference rules of predicate logic, which transform

$$\textit{is-Gröbner-basis}[F]$$

into a truth value using a knowledge base that contains elementary Gröbner bases theory and the above formula (1). (Note that THEOREMA uses brackets for function and predicate application.)

Here is another example of a predicate logic formula (in THEOREMA syntax), which may be read as a (recursive) algorithm:

$$\begin{array}{l}\textit{is-sorted}[\langle\rangle]\\ \forall_x\ \textit{is-sorted}[\langle x\rangle]\\ \forall_{x,y,\overline{z}}\ \textit{is-sorted}[\langle x,y,\overline{z}\rangle] \Leftrightarrow \begin{array}{l} x\le y\\ \textit{is-sorted}[\langle y,\overline{z}\rangle].\end{array}\end{array}$$

(Formulae placed above each other should be read as conjunctions.)

In this algorithmic formula, we use "sequence variables", which in THEOREMA are written as overbarred identifiers: The substitution operator for sequence variables allows the substitution of none, one, or finitely many terms whereas the ordinary substitution operator for the ordinary variables of predicate logic (like x and y in our example) allows only the substitution of exactly one term for a variable. Thus, for example,

$$\textit{is-sorted}[\langle 1,2,2,5,7\rangle],$$

by using the generalized substitution operator for the sequence variable $\overline{z}$, may be rewritten into

$$1 \leq 2$$
$$\wedge$$
$$\mathit{is\text{-}sorted}[\langle 2, 2, 5, 7\rangle],$$

etc.

(The extension of predicate logic by sequence variable is practically useful because it allows the elegant formulation of pattern matching programs. For example, the formula $p[\overline{v}, w, \overline{x}, w, \overline{y}] = 1$ says that p yields 1 if two arguments are identical. The meta-mathematical implications of introducing sequence variables in predicate logic are treated in [16].)

Hence, THEOREMA has the possibility of formulating and executing any mathematical algorithm within the same logic frame in which the correctness of these algorithms and any other mathematical theorem can be proved. In addition, THEOREMA has a possibility to invoke any algorithm from the underlying MATHEMATICA algorithm library as a black box so that, if one wants, the entire functionality of MATHEMATICA can be taken over into the logic frame of THEOREMA.

3 Attractive Syntax

Of course, the internal syntax of mathematical formulae on which algorithmic reasoners may be based, theoretically, is not a big issue. Correspondingly, in logic books, syntax is kept minimal. As a means for human thinking and expressing ideas, however, syntax plays an enormous role. Improving syntax for the formulae appearing in the input and the output of algorithmic reasoners and in mathematical knowledge bases is an important practical means for making future math systems more attractive for working mathematicians. Most of the current reasoning systems, in the past few years, started to add functionalities that allow to use richer syntax.

The THEOREMA system put a particular emphasis on syntax right from the beginning, see the papers on THEOREMA on the home page of the project `http://www.theorema.org/`. We allow two-dimensional syntax and user-programming of syntax, nested presentation of proofs, automated generation of natural language explanatory text in automated proofs, hyperlinks in proofs etc. In recent experiments, we even provided tools for graphical syntax, called "logicographic" syntax, see [19]. The implementation of these feature was made comparatively easy by the tools available in the front-end of MATHEMATICA which is the programming environment for THEOREMA. Of course, whatever syntax is programmed by the user, the formulae in the external syntax is then translated into the internal standard form, which is a nested MATHEMATICA expression, used as the input format of all the THEOREMA reasoners. For example,

$$\forall_{x,y,\overline{z}} \left(\mathit{is\text{-}sorted}[\langle x, y, \overline{z}\rangle] \Leftrightarrow \begin{array}{l} x \leq y \\ \mathit{is\text{-}sorted}[\langle y, \overline{z}\rangle] \end{array} \right)$$

internally is the nested prefix expression:

$$
\begin{array}{l}
{}^{\text{TM}}\text{ForAll}[\\
\quad \bullet\text{range}[\bullet\text{simpleRange}[\bullet\text{var}[x]],\\
\qquad\quad \bullet\text{simpleRange}[\bullet\text{var}[y]],\\
\qquad\quad \bullet\text{simpleRange}[\bullet\text{var}[\bullet\text{seq}[z]]]],\\
\quad \text{True},\\
\quad {}^{\text{TM}}\text{Iff}[\text{is–sorted}[{}^{\text{TM}}\text{Tuple}[\bullet\text{var}[x], \bullet\text{var}[y], \bullet\text{var}[\bullet\text{seq}[z]]]],\\
\qquad\quad {}^{\text{TM}}\text{And}[{}^{\text{TM}}\text{LessEqual}[\bullet\text{var}[x], \bullet\text{var}[y]],\\
\qquad\qquad\quad \text{is–sorted}[{}^{\text{TM}}\text{Tuple}[\bullet\text{var}[y], \bullet\text{var}[\bullet\text{seq}[z]]]]]]]
\end{array}
$$

Translators exist for translating formulae in the syntax of other systems to the THEOREMA syntax and vice versa. A translator to the recent OMDOC [15] standard is under development.

4 Structured Mathematical Knowledge Bases

We think that future math systems need "external" and "internal" tools for structuring mathematical knowledge bases.

External tools are tools that partition collections of formulae into sections, subsections, etc. and maybe, in addition, allow to give key words like 'Theorem', 'Definition' etc. and labels to individual formulae so that one can easily reference and re-arrange individual formulae and blocks of formulae in large collections of formulae. Such tools, which we call "label management tools", are implemented in the latest version of THEOREMA, see [23].

In contrast, internal structuring tools consider the structure of mathematical knowledge bases itself as a mathematical relation, which essentially can be described by "functors" (or, more generally, "relators"). The essence of this functorial approach to structuring mathematical knowledge can be seen in a formula as simple as

$$\forall_{x,y}\ \Big(x \sim y \Leftrightarrow \begin{array}{l} x \le y \\ y \le x \end{array}\Big).$$

In this formula, the predicate $\sim$ is defined in terms of the predicate $\le$. We may want to express this relation between $\sim$ explicitly by defining the higher-order binary predicate AR:

$$\forall_{\sim,\le}\left(AR[\sim,\le] \Leftrightarrow \forall_{x,y}\ \Big(x \sim y \Leftrightarrow \begin{array}{l} x \le y \\ y \le x \end{array}\Big)\right).$$

We may turn this "relator" into a "functor" by defining, implicitly,

$$\forall_{\le}\left(\forall_{x,y}\Big(AR[\le][x,y] \Leftrightarrow \begin{array}{l} x \le y \\ y \le x \end{array}\Big)\right).$$

(In this paper, we do not distinguish the different types of the different variables occurring in formulae because we do not want to overload the presentation with technicalities.)

Functors have a computational and a reasoning aspect. The *computational* aspect of functors already received strong attention in the design of programming languages, see for example [14]: If we know how to compute with $\leq$ then, given the above definition of the functor AF, we also know how to compute with the predicate $AF[\leq]$.

However, in addition, functors have also a *reasoning* aspect which so far has received little attention in reasoning systems: For example, one can easily prove the following "conservation theorem":

$$\forall_{\sim,\leq} \left(\begin{matrix} AR[\sim,\leq] \\ \mathit{is\text{–}transitive}[\leq] \end{matrix} \;\Rightarrow\; \mathit{is\text{–}transitive}[\sim] \right),$$

where

$$\forall_{\leq} \left(\mathit{is\text{–}transitive}[\leq] \;\Leftrightarrow\; \forall_{x,y,z} \left(\begin{matrix} x \leq y \\ y \leq z \end{matrix} \;\Rightarrow\; x \leq z \right) \right).$$

In other words, if we know that $\leq$ is in the "category" of transitive predicates and $\sim$ is related to $\leq$ by the relator AR (or the corresponding functor AF) then $\sim$ also is in the category of transitive predicates. Of course, studying a couple of other conservation theorems for the relator AR, one soon arrives at the following conservation theorem

$$\forall_{\sim,\leq} \left(\begin{matrix} AR[\sim,\leq] \\ \mathit{is\text{–}quasi\text{–}ordering}[\leq] \end{matrix} \;\Rightarrow\; \mathit{is\text{–}equivalence}[\sim] \right),$$

which is the theorem which motivates the consideration of the relator AR.

After some analysis of the propositions proved when building up mathematical theories, it should be clear that, in various disguises, conservation theorems make up a considerable portion of the propositions proved in mathematics.

Functors for computation, in an attractive syntax, are available in Theorema, see for example the case study [6]. Some tools for organizing proofs of conservation theorems in Theorema where implemented in the PhD thesis [24] but are not integrated into the current version of Theorema. An expanded version of these tools will be available in the next version of Theorema.

5 Schemes for Invention

Given a (structured) knowledge base K (i.e. a structured collection of formulae on a couple of notions expressed by predicates and functions), in one step of the theory exploration process, progress can be made in one of the following directions:

- invention of *notions* (i.e. axioms or definitions for new functions or predicates),
- invention and verification of *propositions* about notions,
- invention of *problems* involving notions,

– invention and verification of *methods* (algorithms) for solving problems.

The results of these steps are then used for expanding the current knowledge base by new knowledge. For verifying (proving propositions and proving the correctness of methods), the current reasoning systems provide a big arsenal of algorithmic provers. In the THEOREMA system, by now, we implemented two provers for general predicate logic provers (one based on natural deduction, one based on resolution) and special provers for analysis, for induction over the natural numbers and tuples, for geometric theorems (based on Gröbner bases and other algebraic methods), and for combinatorial identities. We do not go into any more detail on algorithmic proving since this topic is heavily treated in the literature, see the above references on reasoning systems. Rather, in this paper, our emphasis is on algorithmic invention. For this, in this section, we propose the systematic use of formulae schemes whereas, in the next section, we will discuss the use of conjecture generation from failing proofs. In a natural way, these two methods go together in an alternating bottom-up/top-down process.

We think of schemes as formulae that express the accumulated experience of mathematicians for inventing mathematical axioms (in particular definitions), propositions, problems, and methods (in particular algorithms). Schemes should be stored in a (structured) schemes library L. This library could be viewed as part of the current knowledge. However, it is conceptually better to keep the library L of schemes, as a general source of ideas for invention, apart from the current knowledge base K that contains the knowledge that is available on the specific notions (operations, i.e. predicates and functions) of the specific theory to be explored at the given stage.

The essential idea of formulae schemes can, again, be seen already in the simple example of the previous section on functors: Consider the formula

$$\forall_{x,y} \left(x \sim y \Leftrightarrow \begin{array}{c} x \leq y \\ y \leq x \end{array} \right)$$

that expresses a relation between the two predicates $\leq$ and $\sim$. We can make this relation explicit by the definition

$$\forall_{\sim,\leq} \left(AR[\sim, \leq] \Leftrightarrow \forall_{x,y} \left(x \sim y \Leftrightarrow \begin{array}{c} x \leq y \\ y \leq x \end{array} \right) \right).$$

This scheme (which we may also conceive as a "functor") can now be used as a condensed proposal for "inventing" some piece of mathematics depending on how we look at the current exploration situation:

Invention of a new notion (definition, axiom):

If we assume that we are working w.r.t. a knowledge base K in which a binary predicate constant P occurs, then we may apply the above scheme by introducing a new binary predicate constant Q and asserting

$$AR[Q, P].$$

In other words, application of the above scheme "invents the new notion Q together with the explicit definition"

$$\forall_{x,y}\left(Q[x,y] \Leftrightarrow \begin{array}{c} P[x,y] \\ P[y,x] \end{array}\right).$$

Invention of a new proposition:

If we assume that we are working w.r.t. a knowledge base K in which two binary predicate constants P and Q occur, then we may apply the above scheme by conjecturing

$$AR[Q,P],$$

i.e.

$$\left(Q[x,y] \Leftrightarrow \begin{array}{c} P[x,y] \\ P[y,x] \end{array}\right).$$

We may now use a suitable (automated) prover from our prover library and try to prove or disprove this formula. In case the formula can be proved, we may say that the application of the above scheme "invented a new proposition on the known notions P and Q".

Invention of a problem:

Given a binary predicate constant P in the knowledge base K, we may ask to "find" a Q such that

$$AR[Q,P].$$

In this case, application of the scheme AR "invents a problem". The nature of the problem specified by AR depends on what we allow as "solution" Q. If we allow any binary predicate constant occurring in K then the problem is basically a "method retrieval and verification" problem in K: We could consider all or some of the binary predicate constants Q in K as candidates and try to prove/disprove $AR[Q,P]$. However, typically, we will allow the introduction of a new constant Q and ask for the invention of formulae D that "define" Q so that, using K and D, $AR[Q,P]$ can be proved. Depending on which class of formulae we allow for "defining" Q (and possible auxiliary operations), the difficulty of "solving the problem" (i.e. finding Q) will vary drastically. In the simple example above, if we allow to use the given P and explicit definitions, then the problem is trivially solved by the formula $AR[Q,P]$ itself, which can be considered as an "algorithm" w.r.t. the auxiliary operation P. If, however, we allow only the use of certain elementary algorithmic functions in K and only the use of recursive definitions then this problem may become arbitrarily difficult.

Invention of a method (algorithm):

This case is formally identical to the case of invention of an axiom or definition. However methods are normally seen in the context of problems. For example if we have a problem

$$PR[Q,R]$$

of finding an operation Q satisfying PR in relation to certain given operations R then we may try the proposal

$$AR[Q, P]$$

as a method. If we restrict the schemes allowed in the definition of Q (and auxiliary operations) to being recursive equalities then we arrive at the notion of algorithmic methods.

Case Studies:

The creative potential of using schemes, together with failing proof analysis, can only be seen in major case studies. At the moment, within the THEOREMA project, three such case studies are under way: One for Gröbner bases theory, [7], one for teaching elementary analysis, and one for the theory of tuples, [8]. The results are quite promising. Notably, for Gröbner bases theory, we managed to show how the author's algorithm for the construction of Gröbner bases can be automatically synthesized from a problem specification in predicate logic. Since the Gröbner bases algorithm is deemed to be nontrivial, automated synthesis may be considered as a good benchmark for the power of mathematical invention methods.

Not every formula scheme is equally well suited for inventing definitions, propositions, problems, or methods. Here are some examples of typical simple schemes in each of the four areas:

A Typical Definition Scheme:

$$\forall_{P,Q} \left(\mathit{alternating\text{–}quantification}[Q, P] \Leftrightarrow \forall_f \left(Q[f] \Leftrightarrow \forall_x \, \exists_y \, \forall_z \, P[f, x, y, z] \right) \right).$$

Many of the notions in elementary analysis (e.g. "limit") are generated by this (and similar) schemes.

A Typical Proposition Scheme:

$$\forall_{f,g,h} \left(\mathit{is\text{–}homomorphic}[f, g, h] \Leftrightarrow \forall_{x,y} (h[f[x, y]] = g[h[x], h[y]]) \right).$$

Of course, all possible "algebraic" interactions (describable by equalities between various compositions) of functions are candidates for proposition schemes.

A Typical Problem Scheme:

$$\forall_{A,P,Q} \left(\mathit{explicit\text{–}problem}[A, P, Q] \Leftrightarrow \forall_x \begin{array}{l} P[A[x]] \\ Q[x, A[x]] \end{array} \right).$$

This seems to be one of the most popular problem schemes: Find a method A that produces, for any x, a "standardized" form $A[x]$ (that satisfies $P[A[x]]$) such that $A[x]$ is still in some relation (e.g. equivalence) Q with x. (Examples: sorting problem, problem of constructing Gröbner bases, etc.)

Two Typical Algorithm Schemes:

$$\forall_{F,c,s,g,h_1,h_2}\left(\begin{array}{l} \mathit{divide\text{–}and\text{–}conquer}[F,c,s,g,h_1,h_2] \Leftrightarrow \\ \qquad \left(F[x] = \left\{\begin{array}{ll} s[x] & \Leftarrow c[x] \\ g[F[h_1[x]], F[h_2[x]]] & \Leftarrow \text{otherwise} \end{array}\right.\right) \end{array}\right).$$

This is of course the scheme by which, for example, the merge-sort algorithm can be composed from a merge function g, splitting functions h_1, h_2, and operations c, s that handle the base case.

$$\forall_{G,lc,df}\left(\begin{array}{l} \mathit{critical\text{–}pair\text{–}completion}[G,lc,df] \;\Leftrightarrow \\ \left(\begin{array}{l} \forall_F\ (G[F] = G[F, \text{pairs}[F]]) \\ \forall_F\ (G[F,\langle\rangle] = F) \\ \forall_{F,g_1,g_2,\overline{p}} \\ \left(\begin{array}{l} G[F,\langle\langle g_1,g_2\rangle,\overline{p}\rangle] = \\ \text{where}\left[\begin{array}{l} f = lc[g_1,g_2],\ \ h_1 = trd[rd[f,g_1],F],\ \ h_2 = trd[rd[f,g_2],F], \\ \left\{\begin{array}{ll} G[F,\langle\overline{p}\rangle] & \Leftarrow h_1 = h_2 \\ G\big[F \frown df[h_1,h_2], \langle\overline{p}\rangle \asymp \left\langle \langle F_k, df[h_1,h_2]\rangle \mid_{k=1,\ldots,|F|}\right\rangle\big] & \Leftarrow \text{otherwise} \end{array}\right. \end{array}\right. \end{array}\right) \end{array}\right) \end{array}\right).$$

This is the scheme by which, for example, the author's Gröbner bases algorithm can be composed from the function *lc* ("least common reducible", i.e. the least common multiple of the leading power products of two polynomials) and the function *df* ("difference", i.e. the polynomial difference in the polynomial context). The algorithm scheme can be tried in all domains in which we have a reduction function rd (whose iteration is called *trd*, "total reduction") that reduces objects f w.r.t. to finite sets F of objects. The algorithm (scheme) starts with producing all pairs of objects in F and then, for each pair $\langle g_1, g_2\rangle$, checks whether the total reduction of $lc[g_1, g_2]$ w.r.t. g_1 and g_2 yields identical results h_1 and h_2, respectively. If this is not the case, $df[h_1, h_2]$ is added to F.

6 Learning from Failed Reasoning

Learning from failed reasoning can be applied both in the case of proving propositions and in the case of synthesizing methods (algorithms). In this paper, we will sketch the method only for the case of method (algorithm) synthesis.

Let us assume that we are working with some knowledge base K and we are given some problem, e.g.

$$\mathit{explicit\text{-}problem}[A,P,Q],$$

where the predicates P and Q are "known", i.e. they occur in the knowledge base K. For example, P and Q could be the unary predicate *is-finite-Gröbner-basis* and the binary predicate *generate-the-same-ideal*, respectively. (The exact definitions of these predicates are assumed to be part of K. For details see [7]).

Then we can try out various algorithm schemes in our algorithm schemes library L. In the example, let us try out the general scheme *critical-pair-completion*, i.e. let us assume

$$\textit{critical-pair-completion}[A, lc, df].$$

(It is an interesting, not yet undertaken, research subject to try out for this problem systematically all algorithm schemes that are applicable in the context of polynomial domains and study which ones will work. Interestingly, in the literature, so far all methods for constructing Gröbner bases rely on the *critical-pair-completion* scheme in one way or the other and no drastically different method has been found!)

The scheme for the unknown algorithm A involves the two unknown auxiliary functions lc and df. We now start an (automated) proof of the correctness theorem for A, i.e. the theorem

$$\forall_F \left(\begin{array}{l} \textit{is–finite–Gröbner–basis}[A[F]] \\ \textit{generate–the–same–ideal}[F, A[F]] \end{array} \right).$$

Of course, this proof will fail because nothing is yet known about the auxiliary functions lc and df. We now analyze carefully the situation in which the proof fails and try to guess requirements lc and df should satisfy in order to be able to continue with the correctness proof. It turns out that a relatively simple requirements generation techniques suffices for generating, completely automatically, the following requirement for lc

$$\forall_{g1,g2} \left(\begin{array}{l} lp[g1] \mid lc[g1, g2] \\ lp[g2] \mid lc[g1, g2] \\ \forall_p \left(\begin{array}{l} lp[g1] \mid p \\ lp[g2] \mid p \end{array} \Rightarrow (lc[g1, g2] \mid p) \right) \end{array} \right),$$

where $lp[f]$ denotes the leading power product of polynomial f. This is now again an explicit problem specification, this time for the unknown function lc. We could again run another round of our algorithm synthesis procedure using schemes and failing proof analysis for synthesizing lc. However, this time, the problem is easy enough to see that the specification is (only) met by

$$lc[g_1, g_2] = lcm[lp[g_1], lp[g_2]],$$

where $lcm[p, q]$ denotes the least common multiple of power products p and q. In fact, the specification is nothing else then an implicit definition of lcm and we can expect that an algorithm satisfying this specification is part of the knowledge base K.

Heureka! We managed to get the main idea for the construction of Gröbner bases completely automatically by applying algorithm schemes, automated theorem proving, and automated failing proof analysis. Similarly, in a second attempt to complete the proof of the correctness theorem, one is able to derive that, as a possible df, we can take just polynomial difference.

The current requirements generation algorithms from failing proof, roughly, has one rule. Given the failing proof situation, collect all temporary assumptions $T[x_0, \ldots, A[\ldots], \ldots]$ and temporary goals $G[x_0, \ldots m[\ldots, A[\ldots], \ldots]]$, where m is (one of) the auxiliary operations in the algorithm scheme for the unknown algorithm A and x_0, etc. are the current "arbitrary but fixed" constants, and produce the following requirement for m:

$$\forall_{x,\ldots,y,\ldots} (T[x, \ldots, y, \ldots] \Rightarrow G[x, \ldots, m[\ldots, y, \ldots]]).$$

This rule is amazingly powerful as demonstrated by the above nontrivial algorithm synthesis. The invention of failing proof analysis and requirements generation techniques is an important future research topic.

7 Meta-Programming

Meta-programming is a subtle and not widely available language feature. However, we think that it will be of utmost practical importance for the future acceptance of algorithm-supported mathematical theory exploration systems. In meta-programming, one wants to use the logic language both as an object and a meta-language. In fact, in one exploration session, the language level may change several times and we need to provide means for governing this change in future systems. For example, in an exploration session, we may first want to define (by applying the divide–and–conquer scheme) a sorting algorithm

$$\forall_x \left(sort[x] = \begin{cases} x & \Leftarrow \textit{is-short-tuple}[x] \\ merge[x, sort[left[x]], sort[right[x]]] & \Leftarrow \text{otherwise} \end{cases} \right).$$

Then we may want to apply one of the provers in the system to prove the algorithm correct, i.e. we want to prove

$$\forall_x \begin{pmatrix} \textit{is-sorted}[sort[x]] \\ \textit{contain-the-same-elements}[x, sort[x]] \end{pmatrix}$$

by calling, say, a prover *tuple-induction*

$$\forall_x \; \textit{tuple-induction} \left[\begin{pmatrix} \textit{is-sorted}[sort[x]] \\ \textit{contain-the-same-elements}[x, sort[x]] \end{pmatrix}, K_0 \right],$$

and checking whether the result is the constant 'proved', where K_0 is the knowledge base containing the definition of *sort* and *is-sorted* and definitions and propositions on the auxiliary operations.

Now, *tuple-induction* itself is an algorithm which the user may want to define himself or, at least, he might want to inspect and modify the algorithm available in the prover library. This algorithm will have the following structure

$$\begin{array}{l} \textit{tuple-induction}[\forall_x F, K] = \\ \quad \textit{and}[\textit{tuple-induction}[F_{x \leftarrow \langle\rangle}, K], \\ \qquad \textit{tuple-induction}[F_{x \leftarrow \langle x0, \overline{y0} \rangle}, \textit{append}[K, F_{x \leftarrow \langle \overline{y0} \rangle}]], \end{array}$$

where $\leftarrow$ is substitution and x_0 and $\overline{y_0}$ are "arbitrary but fixed" constants (that must be generated from x, F, and K).

Also, *tuple-induction* itself needs a proof that could proceed, roughly, by calling a general predicate logic prover in the following way

$$\textit{general-prover}\ \Big[\forall_{x,F,K}\ \big(\big(\textit{tuple-induction}\big[\forall_x F, K\big] = \text{“}\textit{proved}\text{”}\big) \Rightarrow \big(\textit{append}[\textit{ind}, K] \models \forall_x F\big)\big)\Big],$$

where *ind* are the induction axioms for tuples and $\models$ denotes logic consequence.

Of course, the way it is sketched above, things cannot work: We are mixing language levels here. For example, the '$\forall$' in the previous formula occurs twice but on different language layers and we must not use the same symbol for these two occurrences. Similarly, the '$\forall$' in the definition of tuple-induction has to be different from the '$\forall$' in the definition of *sort*. In fact, in the above sketch of an exploration session, we are migrating through three different language layers.

We could now use separate name spaces for the various language layers. However, this may become awkward. Alternatively, one has to introduce a "quoting mechanism". The problem has been, of course, addressed in logic, see [22] for a recent discussion but we think that still a lot has to be done to make the mechanism practical and attractive for the intended users of future reasoning systems. In Theorema, at present, we are able to define algorithms and formulate theorems, apply algorithms and theorems to concrete inputs ("compute") and prove the correctness of algorithms and theorems in the same session and using the same language, namely the Theorema version of predicate logic. For this, the name spaces are automatically kept separate without any action needed from the side of the user. However, we are not yet at the stage where we could also formulate provers and prove their correctness within the same language and within the same session. This is a major design and implementation goal for the next version of Theorema. An attractive solution may be possible along the lines of [25] and [26].

8 Conclusion

We described mathematical theory exploration as a process that proceeds in a spiral. In each cycle of the spiral, new axioms (in particular definitions), propositions, problems, and methods (in particular algorithms) are introduced and studied. Both invention of axioms, propositions, problems, and methods as well as verification of proposition and methods can be supported by algorithms ("algorithmic reasoners"). For this, at any stage of an exploration, we have to be able to act both on the object level of the formulae (axioms, propositions, problems, methods) and the meta-level of reasoners. We sketched a few requirements future mathematical exploration systems should fulfil in order to become attractive as routine tools for the exploration activity of working mathematicians.

A particular emphasis was put on the interaction between the use of schemes (axiom schemes, proposition schemes, problem schemes, and algorithm schemes)

and the algorithmic generation of conjectures from failing proofs as a general heuristic, computer-supportable strategy for mathematical invention. The potential of this strategy was illustrated by the automated synthesis of the author's Gröbner bases algorithm. The ideas presented in this paper will serve as work plan for the next steps in the development of the THEOREMA system.

References

1. An Interactive Mathematical Proof System. http://imps.mcmaster.ca/.
2. The COQ Proof Assistant. http://coq.inria.fr/.
3. The MIZAR Project. http://www.mizar.org/.
4. The OMEGA System. http://www.ags.uni-sb.de/~omega/.
5. A. Asperti, B. Buchberger, and J. H. Davenport, editors. *Mathematical Knowledge Management: Second International Conference, MKM 2003 Bertinoro, Italy, February 16–18, 2003*, volume 2594 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.
6. B. Buchberger. Groebner Rings in THEOREMA: A Case Study in Functors and Categories. Technical Report 2003-46, SFB (Special Research Area) Scientific Computing, Johannes Kepler University, Linz, Austria, 2003.
7. B. Buchberger. Towards the automated synthesis of a Gröbner bases algorithm. *RACSAM (Review of the Spanish Royal Academy of Sciences)*, 2004. To appear.
8. B. Buchberger and A. Craciun. Algorithm synthesis by lazy thinking: Examples and implementation in THEOREMA. In *Proc. of the Mathematical Knowledge Management Symposium*, volume 93 of *ENTCS*, pages 24–59, Edunburgh, UK, 25–29 November 2003. Elsevier Science.
9. B. Buchberger, C. Dupré, T. Jebelean, F. Kriftner, K. Nakagawa, D. Vasaru, and W. Windsteiger. The THEOREMA project: A progress report. In M. Kerber and M. Kohlhase, editors, *Proc. of Calculemus'2000 Conference*, pages 98–113, St. Andrews, UK, 6–7 August 2000.
10. B. Buchberger, G. Gonnet, and M. Hazewinkel, editors. *Mathematical Knowledge Management: Special Issue of Annals of Mathematics and Artificial Intelligence*, volume 38. Kluwer Academic Publisher, 2003.
11. R. Constable. *Implementing Mathematics Using the* NUPRL *Proof Development System*. Prentice-Hall, 1986.
12. N. G. de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger, editors, *Proc. of Symposium on Automatic Demonstration, Versailles, France*, volume 125 of *LN in Mathematics*, pages 29–61. Springer Verlag, Berlin, 1970.
13. M. Gordon and T. Melham. *Introduction to* HOL*: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.
14. R. Harper. Programming in Standard ML. Carnegie Mellon University. www-2.cs.cmu.edu/~rwh/smlbook/online.pdf, 2001.
15. M. Kohlhase. OMDOC: Towards an internet standard for the administration, distribution and teaching of mathematical knowledge. In J.A. Campbell and E. Roanes-Lozano, editors, *Artificial Intelligence and Symbolic Computation: Proc. of the International Conference AISC'2000*, volume 1930 of *Lecture Notes in Computer Science*, pages 32–52, Madrid, Spain, 2001. Springer Verlag.

16. T. Kutsia and B. Buchberger. Predicate logic with sequence variables and sequence function symbols. In A. Asperti, G. Bancerek, and A. Trybulec, editors, *Proc. of the 3rd Int. Conference on Mathematical Knowledge Management, MKM'04*, volume 3119 of *Lecture Notes in Computer Science*, Bialowieza, Poland, 19–21 September 2004. Springer Verlag. To appear.
17. Zh. Luo and R. Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, 1992.
18. L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs*, volume 806 of *LNCS*, pages 213–237. Springer Verlag, 1994.
19. K. Nakagawa. Supporting user-friendliness in the mathematical software system THEOREMA. Technical Report 02-01, PhD Thesis. RISC, Johannes Kepler University, Linz, Austria, 2002.
20. L. Paulson. ISABELLE: the next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.
21. F. Pfenning. ELF: A meta-language for deductive systems. In A. Bundy, editor, *Proc. of the 12th International Conference on Automated Deduction, CADE'94*, volume 814 of *LNAI*, pages 811–815, Nancy, France, 1995. Springer Verlag.
22. F. Pfenning. Logical frameworks. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 17, pages 1063–1147. Elsevier Science and MIT Press, 2001.
23. F. Piroi and B. Buchberger. An environment for building mathematical knowledge libraries. In A. Asperti, G. Bancerek, and A. Trybulec, editors, *Proc. of the 3rd Int. Conference on Mathematical Knowledge Management, MKM'04*, volume 3119 of *Lecture Notes in Computer Science*, Bialowieza, Poland, 19–21 September 2004. Springer Verlag. To appear.
24. E. Tomuta. An architecture for combining provers and its applications in the THEOREMA system. Technical Report 98-14, PhD Thesis. RISC, Johannes Kepler University, Linz, Austria, 1998.
25. M. Sato. Theory of judgments and derivations. In Arikawa, S. and Shinohara, A. eds., *Progress in Discovery Science*, Lecture Notes in Artificial Intelligence 2281, pp. 78 – 122, Springer, 2002.
26. M. Sato, T. Sakurai, Y. Kameyama and A. Igarashi. Calculi of meta-variables. In Baaz M. and Makowsky, J.A. eds., *Computer Science Logic*, Lecture Notes in Computer Science 2803, pp. 484 – 497, Springer, 2003.