A Survey on the *Theorema* Project*

Bruno Buchberger, Tudor Jebelean, Franz Kriftner,
Mircea Marin, Elena Tomuţa, Daniela Văsaru

Research Institute for Symbolic Computation
A4232 Schloß Hagenberg, Austria

firstname.lastname@risc.uni-linz.ac.at
http://www.risc.uni-linz.ac.at

Abstract

The *Theorema* project aims at *extending current computer algebra systems by facilities for supporting mathematical proving.* The present early-prototype version of the *Theorema* software system is implemented in *Mathematica 3.0.* The system consists of a general higher-order predicate logic prover and a *collection of special provers* that call each other depending on the particular proof situations. The individual provers *imitate the proof style of human mathematicians* and aim at *producing human-readable proofs in natural language* presented in *nested cells* that facilitate studying the computer-generated proofs at various levels of detail. The special provers are *intimately connected with the functors* that build up the various mathematical domains.

1 The Objectives of the *Theorema* Project

The *Theorema* project aims at providing a uniform (logic and software) frame for computing, solving, and proving. In a simplified view, given a "knowledge base" K of formulae (and a logical / computational derivation mechanism L),

- a "*computer*" for K (in an abstract sense) enables the user to provide an expression (term, formula, program) T with a free variable x and a value v (from an appropriate domain) and "evaluates" $T_{x \leftarrow v}$ (T with v substituted for x) w.r.t. K (within the calculus L),
- a "*solver*" for K enables the user to provide an expression T with a free variable x and produces (all) values v for which $T_{x \leftarrow v}$ holds (in L) w.r.t. K, and
- a "*prover*" for K enables the user to provide an expression T with a free variable x and decides whether, for all values v, $T_{x \leftarrow v}$ holds (in L) w.r.t. K.

*This paper was prepared during a stay of the first author as a visiting research fellow at the University of Tsukuba, Japan, Chair of Professor Tetsuo Ida, and was supported by TARA (Tsukuba Advanced Research Alliance), IPA-AITP (Advanced Information Technology Program of the Information-Technology Promotion Agency, Japan) in the frame of the project "Coordination Programming in Open Computing Environment", and by a project grant on "Computer Algebra" by Fujitsu Labs, ISIS, Numazu, Japan.

Of course, computing, solving, and proving are mutually dependent. For example, a prover for a certain class of existentially quantified formulae, essentially, is a solver. Also, certain solvers may need proving potential as a subroutine. Both solvers and provers are expressions that are evaluated on (abstract) computers. Anyway, from the point of view of a user, computers, solvers, and provers are doing three different things with a given expression T. (A more detailed analysis of "computing", "solving", "proving" will be given in another paper.)

One should also remark that, often, what the user expects from a prover is not only a *"black box" decision* about the validity of a given formula T but a *proof*, i.e. a sequence of inference steps (in a certain logic) together with explanations about the assumptions used in each step etc. that derive the validity of T from the knowledge base K.

Existing computer algebra systems are quite good in computing and, for certain K, are also amazingly good in solving (example: the "Solve" routines for systems of non-linear algebraic equations) but weak in proving. Existing theorem proving systems, for certain K (in certain areas of mathematics), are amazingly strong in proving but, usually, are weak in solving and computing. It is natural to strive for systems that are strong in computing, solving, *and* proving.

In order to close the gap between computing / solving (computer algebra systems) on the one side and proving (theorem proving systems) on the other side one could start from either of the two sides. Efforts to close the gap starting from theorem provers are for instance, [9], [15], [18]. Also, there are some recent projects that are based either on adding proving facilities or on linking theorem provers to computer algebra systems, see the bibliography in [13]. The most important current projects combining computer algebra systems with theorem provers (in either of the above approaches) are the following: Isabelle [18], Analytica [7], NQTHM [2], Nuprl [9], Coq [15], HOL [12], Oyster-Clam [14], [20], Redlog [11]. In the *Theorema* project, we decided to start from an existing computer algebra system, namely Mathematica (version 3.0) and to add proving facilities. Our project is different from the other projects in one way or the other. The distinctive features are: integration of an existing rewrite rule language (Mathematica) into our logic; imitation of human proof style; natural language formulation of proofs; combination of functor style programming with proving; integration of proving into math textbooks generation and interactive math teaching; multi-style proving triggered by special context.

In principle, one could of course take any of the existing computer algebra systems as a starting point for our project.

Here are the reasons why, after a profound analysis of the current computer algebra systems, we decided to base our work on Mathematica 3.0 (at least in the prototype phase):

- Essentially, the innermost part of *the Mathematica language is identical to higher-order equational logic* (see the detailed analysis in [5]). As a consequence, Mathematica can be viewed as a "logic-internal" programming language and, thus, the language gap between computing and proving is closed in a natural way.
- The rule-based programming style of Mathematica lends itself to the implementation of provers that are basically a collection of rules for handling proof situations.
- Mathematica provides the *"notebook" facility* by which mathematical papers can be nicely structured and "active text" (programs that can be called from within the paper) can be intermingled with ordinary passive mathematical text.
- The new version 3.0 of Mathematica has highly sophisticated mathematical *typesetting facilities* even for executable formulae in the "input cells" of Mathematica "notebooks". In particular, Mathematica 3.0 also allows to change and extend its syntax and to attach user-defined semantics with syntactical notation.
- The text in notebooks can be structured in *nested cells* of arbitrary nesting depth. Such notebooks can be produced by Mathematica programs (e.g. our provers). This allows an elegant and novel way of presenting proofs in nested form as will be demonstrated below.

Together, these features of Mathematica provide a good starting point for moving towards the final goal of the *Theorema* project, which is to provide a uniform (logic and software) system in which a working mathematician, without leaving the system, can get computer-support while looping through all phases of the mathematical problem solving cycle:

- the phase of *specifying a problem* including the compilation of relevant knowledge and the definition of new concepts (predicates, functions) and auxiliary algorithms;
- the phase of *exploring* a given problem and creating ideas and conjectures by studying examples using the available knowledge and algorithms;
- the phase of *proving* or disproving conjectures and thereby increasing the relevant knowledge base;
- the phase of *programming*, i.e. transforming useful new and provenly correct mathematical knowledge into algorithms for solving the initial problem;
- the phase of *writing up one's finding* in interactive mathematical documents.

The *Theorema* project is pursued at RISC under the direction of the first author. It is an attempt at realizing the goal of integrating computer algebra and theorem proving, which has tentatively been formulated already in [3] and, more explicitly, in [4]. The co-authors of this paper are currently working jointly with the first author on the following parts of the system: syntax of the mathematical language (F. Kriftner), predicate logic prover (T. Jebelean), inductive prover for lists (D. Văsaru), generation of knowledge bases from functors (E. Tomuţa), simplification prover (M. Marin).

2 The Structure of the *Theorema* System

The *Theorema* system is being built up in the following layers:

- *A symbolic computation software system* (at present, Mathematica 3.0) as the basic *implementation frame*. (Note that we do not rely on the algebraic algorithms library of such a system but only on the language frame. More specifically, algorithms should be derived and verified within our system before being integrated into the algorithmic library.)
- *A mathematical language* as a common frame for both non-algorithmic and algorithmic mathematics. Basically, we take a version of higher order logic (with the additional concept of "sequence variables", a concept borrowed from Mathematica for this purpose). The syntax of this language is implemented by using the syntax extension facilities of Mathematica. The part of the language that consists of "executable formulae" (function definitions using induction and bounded quantifiers) gets a semantic interpretation by writing appropriate functions in Mathematica.
- The concept of "*functor*" as the general mechanism for building up towers of mathematical domains. Again, using higher-order variables, the Currying mechanism, and the module concept, functors can be implemented elegantly in Mathematica.
- A general *predicate logic prover* of a "natural style" introduced in earlier papers by Buchberger, see for example [6], implemented in Mathematica.
- Various *special theorem provers (and / or interactive proof developers) corresponding, in a natural way, to the various functors* that build up mathematical domains. The design and implementation of these provers is the present main priority in the *Theorema* project. All these provers produce proofs that imitate "natural" proof styles of human mathematicians. By now, an induction prover for the natural numbers and one over the domain of lists over a given domain are already implemented, a prover for the domain of multivariate polynomials over a given domain of monomials is under way.
- Various special *black-box theorem provers* that decide the validity of theorems in certain special areas (like the theory of real closed fields or certain geometrical theorems) by reduction of the proof problem to certain algebraic problems solved by purely algebraic algorithms (Groebner bases method, see [16]; characteristic sets method, see [22]; cylindrical algebraic decomposition, see [8]; Cayley factorization, see [19]).
- Various *special solvers* including the ones already available in the current symbolic computation systems (notably the solvers for multivariate polynomial equation systems).

- A general facility that allows the *presentation of proofs in natural language and in the form of "nested cells"*, which is crucial for being able to read complicated proofs without losing the overview. This facility is based on the facilities for manipulating cells in Mathematica notebooks.

- Mechanisms for the *automatic generation of complicated knowledge bases* from algebraic properties of given domains and the definition of functors.

In the present paper we demonstrate the following parts of the system by way of examples:

– The syntax of the higher order predicate logic language and the semantics of its "executable" part, Section 3.
– The general predicate logic prover, Section 4.
– The induction provers for natural numbers and lists, based on simplification, Section 5.
– The syntax and semantics of the functor concept and computing and proving with functors, Section 6.

These parts of the system are already implemented and extensive experiments are being carried out.

3 A Mathematical Language

The language in which we express both mathematical statements and algorithms is a version of the language of higher order predicate logic. This language starts from *object symbols* like "a", "Sin", "2", etc. (Which can be used either as constants, bound variables or free variables, see below.) The essential construct is *function application* by which terms are built inductively from object symbols using the following two rules:

- Every object symbol is a term.
- If T and $T_1, \ldots, T_n$ are terms, then $T[T_1, \ldots, T_n]$ is a term.

Note that we do not fix the "arity" of object symbols and, in this stage, we do not define any type restrictions. (Type restrictions will be necessary, however, for formulating some of the inference rules). Thus, $Sin[2]$, $2[Sin]$, and $Sin[2,3]$ all are admissible terms. Also, note that "Currying" is possible i.e, for example, $Polys[D][+][5,7]$ is an admissible term. We use "$\equiv$ " as equality symbol.

Furthermore, we allow the usual propositional connectives $\neg$, $\wedge$, etc. and the quantifiers $\exists$ and $\forall$ (with the bound variables written under the quantifiers) for building up formulae from terms. (By quantification, object symbols become bound variables!) For example,

$$\forall_x \exists_y\, f[x,y] \Rightarrow g[x,y]$$

is a typical formula.

Some of the object symbols may be written infix or pre-postfix. For example, $|x+y|$ is a term, in which $+$ is written infix and $||$ is written pre-postfix.

Next we introduce sets, tuples and (natural, integer, and rational) numbers as basic categories of objects with a couple of special operations and quantifiers on these objects. (In distinction to the usual convention in Mathematica, we use curly brackets for the operation of explicit set enumeration and angle brackets for the operation of explicit tuple construction.) Here we give only a few examples of (true) formulae involving these special symbols:

$$\begin{aligned}
\{2,3,4,3\} &\equiv \{3,2,4\} \\
|\{2,3,4,3\}| &\equiv 3 \\
\{x | x \neq x\} &\equiv \{\} \\
\langle 2i \underset{i=2,\ldots,10}{|} prime[i]\rangle &\equiv \langle 4,6,10,14\rangle \\
\langle 2,4,6,10,14\rangle_3 &\equiv 6
\end{aligned}$$

In quantified formulae we reserve two lines for special information on the bound variables: The first line (as subscript) indicates the range of the bounded variable, whereas the second line (as underscript; may also be empty) describes a condition on the bounded variable. For example, in

$$\underset{prime[x]}{\forall_{x \in A}}\quad x \geq n$$

$x \in A$ specifies a range, whereas $prime[x]$ specifies a condition. This distinction (which is normally not made in logic) is important for those formulae that are "executable": The range specification describes a routine by which (finitely many) objects are enumerated, the condition specifies a routine by which, for each object enumerated, it is decided whether or not it should be considered in the body of the quantified formula. For example, if A is a finite set, the above formula can be read as an algorithm.

We implemented various general and special quantifiers like: "$\forall$", "$\exists$", set braces "$\{\}$", summation "Σ", product "Π", etc. which all can be formulated in terms of "λ" which is available as a standard construct in Mathematica. (In fact, in [10] it was shown that the availability of "λ" is crucial.)

Furthermore, we introduce a special category of symbols, called "sequence symbols". (This is a useful concept borrowed from Mathematica where the notation with three underscores is used.) We use an overbar for denoting sequence symbols. For example, $f[\overline{x}, 0, \overline{y}]$ is a term involving the sequence constants $\overline{x}$ and $\overline{y}$. Whereas ordinary constants denote individual objects, sequence constants denote finite ordered sequences of objects. Also sequence symbols can be used as sequence constants, free sequence variables or bound sequence variables, see below.

Sequence constants (and sequence variables occurring as bounded variables in quantified formulae) need particular inference rules. We present here only one such rule, which is a general induction principle on sequence variables:

In order to prove that $\forall_{\overline{x}} A$, where A contains $\overline{x}$ unquantified, proceed as follows:

- *Induction base*: Prove $A_{\overline{x}\leftarrow}$ (i.e. the formula that results from A by replacing $\overline{x}$ by nothing).
- *Induction hypothesis*: Choose $\dot{x}_0$ and $\overline{x_0}$ "arbitrary but fixed" (i.e. introduce a fresh object symbol $\dot{x}_0$ and a fresh sequence symbol $\overline{x_0}$) and assume $A_{\overline{x}\leftarrow\overline{x_0}}$.
- *Induction step:* Prove $A_{\overline{x}\leftarrow\dot{x}_0,\overline{x_0}}$.

For understanding this rule, one must specify the substitution process for sequence symbols. We explain this in an example where the above principle specializes to an induction principle over lists. Let us consider the following formula:

$$\forall_{\overline{x}}\ (a \smile \langle\overline{x}\rangle) \frown b\ \equiv \langle a, \overline{x}, b\rangle,$$

where $\smile$ and $\frown$ denote the operations of prepending, respectively appending, to a list. We have (when doing induction over $\overline{x}$):

– induction base:	$(a \smile \langle\rangle) \frown b \equiv \langle a, b\rangle$,
– induction hypothesis:	$(a \smile \langle\overline{x_0}\rangle) \frown b \equiv \langle a, \overline{x_0}, b\rangle$,
– induction step:	$(a \smile \langle\dot{x}_0, \overline{x_0}\rangle) \frown b \equiv \langle a, \dot{x}_0, \overline{x_0}, b\rangle$.

Using the above induction principle one can produce induction principles for inductively defined domains as shown in the section on induction provers for natural numbers and for lists.

There are two classes of formulae that are "executable":

- inductive equalities,
- formulae using only quantifiers with bounded ranges.

Bounded ranges are finite sets (we saw an example of this above), ranges of integers specified by an upper and lower bound (see the example above with $i = 2, \ldots, n$), and ranges in enumerable domains (generated by functors) specified by an upper and lower bound. (We cannot go into the details of general bounded domain ranges in this paper.)

An example of an inductive function definition by equalities using sequence symbols is the following:

$$\begin{array}{rcl} map[f\_, \langle\rangle] & := & \langle\rangle; \\ map[f\_, \langle obj\_, \overline{objs}\_\rangle] & := & f[obj] \smile map[f, \langle objs\rangle]. \end{array}$$

Here, instead of using a universal quantifier for quantifying the ordinary object symbols f and obj and the sequence symbol $\overline{objs}$, we declared these symbols as universally quantified (or, equivalently, as "free variables") by writing an underscore immediately to the right of the symbols. This is in correspondence with the Mathematica convention for declaring variables. Note that this declaration is only done on the left-hand side of the equalities. Note also that all symbols not explicitly declared to be free or bound variables are constants.

The inductive function definitions and quantified formulae with bounded ranges constitute a sublanguage of our variant of higher order predicate logic. This sublanguage is in fact a universal programming language. Except for notional differences, the inductive definitions part of this language is identical to the essential kernel of the Mathematica language. The bounded quantifier part can be easily implemented by writing a few routines in Mathematica. Hence, an interpreter for this sublanguage is readily available within the Mathematica system.

4 The Predicate Logic Prover

This part of the system handles general proof situations consisting of general (higher-order) predicate logic formulae. We remark that we do not strive for logical completeness, rather we emphasize producing "natural proofs" for important and frequent proof situations with a variety of different proof rules.

Our predicate logic prover handles "*proof situations*" that, roughly, consist of a "goal" (the predicate logic formula to be proved) and a "knowledge base" (predicate logic formulae used as assumptions). For reference purpose, we provide *labels* for all formulae in the initial knowledge base and goal and we generate new labels systematically for all intermediate formulae. The following pair represents a concrete proof situation, where the goal and its label are the first element of the pair while the second element of the pair is the list of assumptions, each consisting of a label and a formula:

$\langle$ "AC", $A \Rightarrow C$ $\rangle$, $\langle$ $\langle$ "AB", $A \Rightarrow B$ $\rangle$, $\langle$ "BC", $B \Rightarrow C$ $\rangle$ $\rangle$

In one proof step, the prover proceeds from a proof situation to one or more new proof situations by using various inference rules. The *inference rule* to be applied in a given proof situation is triggered by the outermost symbols of the formulae constituting the goal and the current knowledge base. Thus, basically, our prover is a "sequent calculus". However, striving for "natural" and easy-to-read proofs, we formulate and use many more rules than the ones that are usually compiled in the calculi in logic texts (which strive for minimality). These rules are taken from [6] and are the ones the first author also uses in his proof training courses for graduate students. A typical proof rule of this kind is the "deduction rule" (for proving $F_L \Rightarrow F_R$, assume F_L and prove F_R):

Replace: $\langle label, F_L \Rightarrow F_R\rangle$, $\langle\langle l_1, A_1, \rangle, \langle l_2, A_2, \rangle, \ldots\rangle$
By: $\langle label_R, F_R\rangle$, $\langle\langle label_L, F_L\rangle, \langle l_1, A_1\rangle, \langle l_2, A_2\rangle, \ldots\rangle$

We now briefly describe the intermediate data structure we use in our prover: Starting from an initial proof situation, we produce a "*proof object*" containing this proof situation. In each proof step this proof object is expanded until we, finally, obtain a proof object that represents the proof, i.e. it contains all the information on the intermediate proof situations and the inference rules used in each step of the proof. The proof object is an abstract object that is not meant to be read by humans. However, it contains all the information necessary to produce, in a second step, a proof in natural language and well structured in "nested cells", see below. In more detail, a proof object can have one of the following two forms:

a) PND[*formula, knowledge-base*]
is an unevaluated proof object (PND stands for "Proof by Natural Deduction"), containing only the proof situation: *formula* is the labeled formula to be proved, and *knowledge-base* is the current knowledge base (axioms, definitions, temporary assumptions, etc.) expressed as a tuple of labeled formulae. Since the prover is a collection of rewrite rules of the form:

PND[*proof-situation*] := *proof-object*,

Mathematica internal evaluation engine will always try to replace an unevaluated proof object by applying one of the rules – this is how the proving engine works.

b) $\langle$ *proof-rule-info, tuple-of-proof-objects, proof-result* $\rangle$
is a [partially] evaluated proof object, where:

- *proof-rule-info*: information on the prover (proof rule) applied in the current proof situation; this information comprises the name of the prover (proof rule), the (labels of the) formulae involved in this proof situation, and new formulae formed in this proof situation,
- *tuple-of-proof-objects*: the tuple of proof objects describing the subproofs generated by applying the prover (proof rule) in the current proof situation – while the proof is not yet completed some of these proof-objects (or sub-objects of them) are still unevaluated. The *tuple-of-proof-objects* can also be empty: in this case we have a terminal proof object which indicates the termination of a branch of the proof.
- *proof-result*: information on whether the proof was successful or not. This element is missing if the proof object contains still unevaluated sub-objects. However, the *proof-result* is always present in a terminal proof object. Also, the mechanism of composing sub-proofs (which we do not describe here for lack of space) will insure that the proof results of the various proof objects are composed correctly and inserted as the last element of the final proof object.

In the example of the inference rule above, the proof object constructed by applying the rule has the following form:

$\langle$ $\langle$ "deduction rule", $label, label_R, F_R, label_L, F_L\rangle$,
PND$[\langle label_R, F_R\rangle$, $\langle\langle label_L, F_L\rangle$, $\langle l_1, A_1\rangle$, $\langle l_2, A_2\rangle$, $\ldots\rangle]\rangle$

If the proof succeeds, the final proof object will have the form:

$\langle$ $\langle$ "deduction rule", $label, label_R, F_R, label_L, F_L\rangle$,
proof-object,
"proved"$\rangle$

where *proof-object* contains the proof of F_R.

Our notion of proof object is an ad-hoc notion which is sufficient for the practical purpose of our system. In a later stage of our project, we will study how these proof objects can be translated into the formal proof objects described in [1] (p. 202), in the frame of the Curry-Howard formalism.

After having produced the proof object, our prover enters into the stage of producing *the natural language version of the proof in nested cell representation.* This is done by syntactical transformation of the proof object into a proof expressed in human style which explains in detail each step of the proof in a structured manner. Failed proofs will also be displayed in this way, giving precise information about the place of failure: this is an important advantage over other "black-box" automatic provers, because it helps the mathematician to locate possible errors in the formulation of the problem.

The main programming tool we use in this post-processor is the `NotebookPut` command of Mathematica with the Option `CellGrouping -> Manual`. By this, a notebook is generated that consists of the nested Mathematica cells produced by our program. We programmed a couple of auxiliary functions for generating certain standard types of cells (labeled formula cells, proof text cells, etc.). By setting the option `CellGrouping -> Manual`, the built-in styles for nested cells are replaced by our own styles and the nesting can be of "arbitrary" depth ("arbitrary" := to a depth that is limited by the width of the screen.) As soon as the notebook with the nested cells representing the proof is generated the user can click the desired parts of the proof "open" or "close" as described in the Mathematica manual [21]: By the recursive structure of the proof objects and the corresponding recursive call of the function "`Nested-Presentation`", entire subproofs may be compressed (by clicking the cell containing the subproof "closed") to only one line when studying a proof. Thus, the reader of a proof may decide himself to which level of detail he wants to study a proof.

The natural language text for each proof rule can be composed freely by the "prover programmer". In the particular implementation, emphasis is put on logically and syntactically correct text. The text produced has a quality that can compete with proof text produced by good students. Of course, it may sometimes read a little boring. However, the principle is: "Better boring but correct than poetic and incorrect." In Figure 1 we give an example of a proof which is generated automatically by our prover (including the natural language text and the cell brackets at the right-hand side, which represent the nested cell structure).

5 Induction Provers

In this section we describe two special provers which are already implemented. Both can prove theorems of the following kind:

$$\forall_{x,y,\ldots}\ rhs \equiv lhs$$

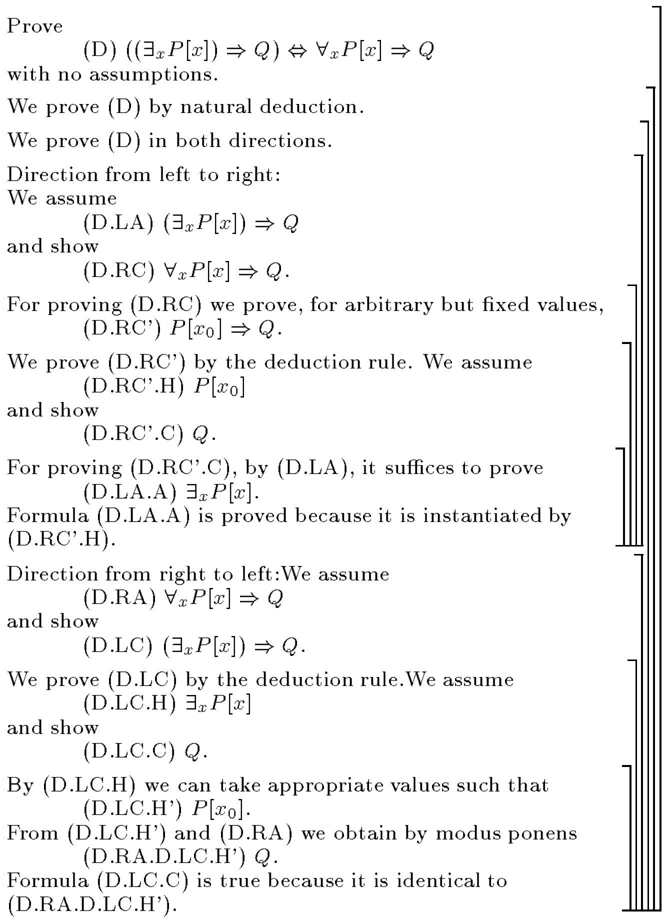
where rhs and lhs are terms containing functions inductively defined by equalities in either the domain of natural numbers in the representation 0, $s[0]$, $s[s[0]]$,... (we will use x^+ for $s[x]$) or in the domain of lists (over some domain) in the representation $\langle\rangle$, $\langle a_1\rangle$, $\langle a_1, a_2\rangle$,.... We could define these domains by functors, see next section. However, in order not to overload this presentation, we just assume that these domains are "given". In these provers, the knowledge base consists of the inductive equalities that define the function symbols occurring in lhs and rhs. A typical definition of this kind is the definition of $\oplus$(plus) on the natural numbers:

$$\begin{array}{lrcl} \forall_m & m\oplus 0 & \equiv & m, \\ \forall_{m,n} & m\oplus n^+ & \equiv & (m\oplus n)^+, \end{array}$$

or the definition of $||$ (length) on the domain of lists:

$$\begin{array}{lrcl} & |\langle\rangle| & \equiv & 0, \\ \forall_{x,\overline{y}} & |\langle x,\overline{y}\rangle| & \equiv & 0^+\oplus|\langle\overline{y}\rangle|. \end{array}$$

The provers proceed recursively over the number n of universally quantified variables: First, all variables are replaced by "arbitrary but fixed" constants and a proof by simplification is attempted (see below). If the formula cannot be proved by simplification then we use induction over the first variable. The base case and the induction step become proofs of


Prove
(D) $((\exists_x P[x]) \Rightarrow Q) \Leftrightarrow \forall_x P[x] \Rightarrow Q$
with no assumptions.

We prove (D) by natural deduction.

We prove (D) in both directions.

Direction from left to right:
We assume
(D.LA) $(\exists_x P[x]) \Rightarrow Q$
and show
(D.RC) $\forall_x P[x] \Rightarrow Q$.

For proving (D.RC) we prove, for arbitrary but fixed values,
(D.RC') $P[x_0] \Rightarrow Q$.

We prove (D.RC') by the deduction rule. We assume
(D.RC'.H) $P[x_0]$
and show
(D.RC'.C) Q.

For proving (D.RC'.C), by (D.LA), it suffices to prove
(D.LA.A) $\exists_x P[x]$.
Formula (D.LA.A) is proved because it is instantiated by (D.RC'.H).

Direction from right to left:We assume
(D.RA) $\forall_x P[x] \Rightarrow Q$
and show
(D.LC) $(\exists_x P[x]) \Rightarrow Q$.

We prove (D.LC) by the deduction rule.We assume
(D.LC.H) $\exists_x P[x]$
and show
(D.LC.C) Q.

By (D.LC.H) we can take appropriate values such that
(D.LC.H') $P[x_0]$.
From (D.LC.H') and (D.RA) we obtain by modus ponens
(D.RA.D.LC.H') Q.
Formula (D.LC.C) is true because it is identical to (D.RA.D.LC.H').

Figure 1: Proof generated automatically.

formulae with one variable less! For these proofs, we call our prover recursively. The induction principle for the natural numbers is straight-forward (base case: $n \leftarrow 0$; induction hypothesis: $n \leftarrow n_0$; induction step: $n \leftarrow (n_0)^+$). The induction principle for lists is an immediate application of the induction principle for sequence symbols introduced in Section 3.

For the sub-proofs by simplification, one could use, in principle, the built-in simplifier of Mathematica. However, for more subtle proofs, we need to have more control over the simplification process and also to trace it step-by-step. Therefore, we implemented our own simplifier.

We give here an example proof automatically generated by the induction prover over lists. The cells corresponding to simplification are closed.

PROPOSITION:

$$(|\asymp|) \quad \forall_{\overline{x},\overline{y}} \quad |\langle\overline{x}\rangle \asymp \langle\overline{y}\rangle| \equiv |\langle\overline{x}\rangle| \oplus |\langle\overline{y}\rangle|$$

under the assumptions

$$\begin{array}{rlrcl}
(\asymp) & \forall_{\overline{x},\overline{y}} & (\langle\overline{x}\rangle \asymp \langle\overline{y}\rangle) & \equiv & \langle\overline{x},\overline{y}\rangle, \\
(|\langle\rangle|) & & |\langle\rangle| & \equiv & 0, \\
(|\langle\cdot,\ldots\rangle|) & \forall_{x,\overline{y}} & |\langle x,\overline{y}\rangle| & \equiv & 0^+ \oplus |\langle\overline{y}\rangle|, \\
(0\oplus) & \forall_{n} & 0 \oplus n & \equiv & n, \\
(+\oplus) & \forall_{m,n} & m^+ \oplus n & \equiv & (m \oplus n)^+.
\end{array}$$

PROOF:
We try to prove

$$(|\asymp|.V) \quad \forall_{\overline{x},\overline{y}} \quad |\langle\overline{x}\rangle \asymp \langle\overline{y}\rangle| \equiv |\langle\overline{x}\rangle| \oplus |\langle\overline{y}\rangle|$$

by simplification. For this, we take all variables arbitrary, but fixed and try to prove

$$(|\asymp|.V.F) \qquad |\langle\overline{x_0}\rangle \asymp \langle\overline{y_0}\rangle| \equiv |\langle\overline{x_0}\rangle| \oplus |\langle\overline{y_0}\rangle|.$$

An attempt for proving this by simplification fails.
Now we prove the simplified formula

$$(|\asymp|.V.S) \quad \forall_{\overline{x},\overline{y}} \quad |\langle\overline{x},\overline{y}\rangle| \equiv |\langle\overline{x}\rangle| \oplus |\langle\overline{y}\rangle|$$

by induction on $\overline{x}$.
Induction base for $\overline{x}$:
We prove

$$(|\asymp|.V.IB) \quad \forall_{\overline{y}} \quad |\langle\overline{y}\rangle| \equiv |\langle\rangle| \oplus |\langle\overline{y}\rangle|.$$

For this, we take in $(|\asymp|.V.IB)$ all variables arbitrary, but fixed and prove

$$(|\asymp|.V.IB.F) \qquad |\langle\overline{y_0}\rangle| \equiv |\langle\rangle| \oplus |\langle\overline{y_0}\rangle|.$$

A proof by simplification works.
Induction hypothesis for $\overline{x}$:

$$(|\asymp|.V.IH) \quad \forall_{\overline{y}} \quad |\langle\overline{x_0},\overline{y}\rangle| \equiv |\langle\overline{x_0}\rangle| \oplus |\langle\overline{y}\rangle|.$$

Induction step for $\overline{x}$:
We prove

$$(|\asymp|.V.IS) \quad \forall_{\overline{y}} \quad |\langle\dot{x}_0,\overline{x_0},\overline{y}\rangle| \equiv |\langle\dot{x}_0,\overline{x_0}\rangle| \oplus |\langle\overline{y}\rangle|.$$

For this, we take in $(|\asymp|.V.IS)$ all variables arbitrary, but fixed and prove

$$(|\asymp|.V.IS.F) \qquad |\langle\dot{x}_0,\overline{x_0},\overline{y_0}\rangle| \equiv |\langle\dot{x}_0,\overline{x_0}\rangle| \oplus |\langle\overline{y_0}\rangle|.$$

A proof by simplification works.

---

6 Functors

6.1 The Concept of Functor in *Theorema*

Functors are a well known concept for building up mathematics in a structured way. In the frame of the *Theorema* project, we adopt the notion of functor in the sense of ML (see [17]), which is technically slightly different from the notion of a functor in the sense of category theory. It is important to observe that, although not explicitly mentioned in the Mathematica manual, functors can easily be implemented in Mathematica. We will publish the details of our approach for implementing functors in Mathematica in a different paper in order not to overload this overview on the *Theorema* project. Here, we only summarize the main idea of the approach.

However, we would like to *emphasize the role functors can play in structuring special provers* of the type described in this paper. We believe that, together with each functor implemented for building up certain new domains from given domains, one should also provide theorem provers that specialize in proving theorems about the domains generated by the functor. In other words, we consider a functor not only as a mechanism for defining a new *domain* (collection of operations) in terms of a given domain but also as a mechanism for transporting *knowledge* about the given domain to the new domain. The problem of proving this kind of "transport knowledge" properties is addressed in the next section where we also give an example of an automatically generated proof.

6.2 Computing with Functors

In this section we briefly summarize the way we implement functors in Mathematica. We start with representing a domain D as a function that defines functions for certain "operators". For example, the following object Z is a simple domain:

$$Z[+] := \mathit{Plus}, \quad Z[0] := 0,$$

where "Plus" and "0" are the "built-in" addition and the built-in integer of Mathematica. For syntactic simplicity, we introduce the convention that " $\underset{D}{o}$ " etc. stands for "$D[o]$" (which is perfectly possible in the new version of Mathematica). Thus, we could write, for example, $2 \underset{Z}{+} 2$ which, with the above definitions, evaluates to 4.

We also add the unary operator "ϵ" to any domain with the convention that, for any domain D, $D[\epsilon]$ is a decision function which yields "True" for exactly the objects we consider to be in the "carrier" of D. For example, we could define

$$Z[\epsilon] := \text{IntegerQ}$$

where "IntegerQ" is the built-in decision algorithm for integers. Note that the symbol "ϵ" is different from the symbol "$\in$" that denotes the element predicate of set theory.

In our terminology (which is basically the ML terminology), a functor is just a function that maps domains into domains. Thus, a functor is an object F that takes D as an argument and produces $F[D]$ with the view that $F[D]$ can now be applied to any operation symbol o yielding an operation in the domain $F[D]$. In particular, $F[D][\epsilon]$ is a decision function for the objects which we want to be in the carrier of $F[D]$.

For constructing functors we provide a function $\mathit{Functor}$. The function parses the *Theorema* formulae passed as arguments, performs a basic correctness check on them, transforms them into Mathematica rules and wraps them in a *Module* construct. In other words the function produces an internal representation of the functor definition which captures the computational content of the formulae input by the user and is directly executable by Mathematica.

With this function one can define the operations in $F[D]$ depending on D as shown in the following simple example that generates the complexification C of a domain D.

$$\text{Complexification}[D\_] := \text{Functor}[\langle C, \epsilon, o, +, \ldots\rangle,$$

$$\langle "\epsilon", \forall_{r,i} \underset{C}{\epsilon}[\langle r, i\rangle] :\Leftrightarrow \underset{D}{\epsilon}[r] \wedge \underset{D}{\epsilon}[i]\rangle;$$

$$\langle "o", \underset{C}{o} \equiv \langle \underset{D}{o}, \underset{D}{o}\rangle\rangle;$$

$$\langle "+", \forall_{xr,xi,yr,yi} \langle xr, xi\rangle \underset{C}{+} \langle yr, yi\rangle \equiv \langle xr \underset{D}{+} yr, xi \underset{D}{+} yi\rangle\rangle;$$

$$\vdots$$

$$C];$$

The function *DefineDomain* is provided for defining new domains in terms of existing functors and domains. For example, for defining the domain G obtained by applying *Complexification* to Z, we can enter:

$$G \text{ ~}DefineDomain\text{~ } Complexification[Z].$$

The function *DefineDomain* can be seen as a specialized assignment used in domain definitions in place of the usual assignment (:=) of Mathematica. Its task is twofold:
– it stores "type" information related to the defined symbol in an internal *Theorema* structure, making it possible to "reason" about the symbol;
– it executes the representation of the functor definition produced by the function *Functor* (see above) associating Mathematica rules to the new domain which is being constructed.
After defining G as shown above, one can compute, for example, $\langle 2, 3\rangle \underset{G}{+} \langle 4, 6\rangle$, which yields $\langle 6, 9\rangle$.

6.3 Proving Theorems about Functors

We address the problem of proving statements of the form $P[D]$ under the assumptions: $D = F[D_1, \ldots, D_k]$, $P_1[D_1]$, $\ldots P_k[D_k]$, where F is a functor; D, D_1, …, D_k are domains; $P, P_1, \ldots, P_k$, are sets of *Theorema* formulae expressing properties of the respective domains. For instance, such a $P[D]$ can contain the axioms of a group:

$$\text{isGroup}[D\_] :=$$

$$\langle\langle "\text{assoc}", \forall_{x,y,z} (x \underset{D}{+} y) \underset{D}{+} z \equiv x \underset{D}{+} (y \underset{D}{+} z)\rangle,$$

$$\langle "\text{zero}", \forall_x \underset{D}{o} \underset{D}{+} x \equiv x\rangle,$$

$$\langle "\text{inverse}", \forall_x x \underset{D}{+} (\underset{D}{-} x) \equiv \underset{D}{o}\rangle$$

$$\rangle;$$

Such statements are special cases of general formulae involving functors and thus are a good example for studying and extending functors. In particular they demonstrate the transfer of knowledge from existing domains to a newly constructed one, a reasoning pattern which is very often encountered in "real life" mathematics.

In *Theorema*, the proof of a statement of the kind shown above consists in the following main steps:

Generation of the proof situation: The formulae of the form $P_i[D_i]$ are replaced with the sequence of formulae defining them, while formulae of the form $D = F[D_1, D_2, \ldots, D_k]$ are replaced with a list of formulae extracted from the definition of the functor F with the input and output domain variables properly instantiated.

Natural Deduction Steps: The logical connectors and the free variables are eliminated (instantiated) by a applying natural deduction inference rules.

Call Specialized Provers: Typically, the problem is to reduce the predicates (equalities) expressed over constants in the constructed domain to predicates expressed in the base domains. For this, special provers are called (e.g. simplifiers, induction provers) which correspond to the structure of the functor.

In our current implementation the steps above are performed by a dedicated prover called *FND* (Functor Natural Deduction). The prover produces a *Theorema* proof-object and provides an option for formatting the proof-object as a Mathematica 3.0 notebook. The proof-objects returned by the specialized provers are embedded into the main proof-object.

We illustrate the way the functor prover works, by listing below the proof automatically generated by the *Theorema* system for the complexification of a group. (For reasons of space, only the proof of the second subgoal is given in full detail.) Note the special sub-proof by simplification, which applies the adequate known equalities as rewrite rules in order to reduce the properties of G to properties of Z.

Prove: (G) $IsGroup(G)$
under the assumptions:
(C) $G \equiv Complexification(Z)$
(Z) $IsGroup(Z)$.

Proof:
We prove the individual conjunctive parts of (G):

□ Proof of
(G.assoc) $\forall_{x,y,z} (x \underset{G}{+} y) \underset{G}{+} z \equiv x \underset{G}{+} (y \underset{G}{+} z)$.

...

□ Proof of
(G.zero) $\forall_x \underset{G}{o} \underset{G}{+} x \equiv x$.

We prove, for arbitrary but fixed values,
(G.zero') $\underset{G}{o} \underset{G}{+} x_0 \equiv x_0$.

By (C.ϵ) we can take appropriate values in Z such that:
(C.ϵ') $x_0 \equiv \langle r_0, i_0\rangle$.

A proof of (G.zero') by simplification works.
Simplification of the LHS term:

$\underset{G}{o} \underset{G}{+} x_0 \equiv$ by (C.ϵ'.1)

$\underset{G}{o} \underset{G}{+} \langle r_0, i_0\rangle \equiv$ by (C.o)

$\langle \underset{Z}{o}, \underset{Z}{o}\rangle \underset{G}{+} \langle r_0, i_0\rangle \equiv$ by (C.+)

$\langle \underset{Z}{o} \underset{Z}{+} r_0, \underset{Z}{o} \underset{Z}{+} i_0\rangle \equiv$ by (Z.zero)

$\langle r_0, \underset{Z}{o} \underset{Z}{+} i_0\rangle \equiv$ by (Z.zero)

$\langle r_0, i_0\rangle$⌋

Simplification of the RHS term:
$x_0 \equiv$ by (C.ϵ'.1)
$\langle r_0, i_0 \rangle$]

□ Proof of
(+.inverse.left) $\forall_x \ x \underset{G}{+} (\underset{G}{-} x) \equiv \underset{G}{o}$.

...

---

7 Conclusion

We described the objectives and present state of the *Theorema* project. Although this project is in its initial stage, we believe that it demonstrates a high potential for enhancing the current computer algebra systems by computer-support for the proving phase of the mathematical problem solving cycle. Our next step will be the systematic design and implementation of special provers for all the fundamental domains and functors, which are used in building up (algorithmic) mathematics. Our first priority is to design provers that generate easy-to read proofs in natural language. Later, as part of our project, we will also re-implement or link the existing "black-box" provers based on algebraic algorithms, to our system. Also, we will integrate special solvers for the various domains and functors.

References

[1] Barendregt, H. Lambda Calculi with Types. In *Handbook of Logic in Computer Science*, Abramsky, Gabbai, and Maibaum, Eds., vol. II. Oxford University Press, 1992, pp. 117–309.

[2] Boyer, R. S., and Moore, J. S. *A Computational Logic Handbook*, vol. 23 of *Perspectives in computing*. Academic Press Inc., 1988.

[3] Buchberger, B. Editorial. *Journal of Symbolic Computation 1*, 1 (1985), 1–6.

[4] 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.

[5] Buchberger, B. Mathematica as a Rewrite Language. In *Proceedings of the Fuji Conference on Functional Logic Programming, Shonan Village, Nov 1-4, 1996* (1996), T. Ida, Ed., Telos publishing conference. To appear.

[6] Buchberger, B., and Lichtenberger, F. *Mathematics for Computer Science I - The Method of Mathematics (German)*. Springer, Berlin, Heidelberg, New York, 1981. Second Edition.

[7] Clarke, E. M., and Zhao, X. Analytica - A theorem prover for Mathematica. *The Mathematica Journal 3*, 1 (1993), 56–71.

[8] 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.

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

[10] de Bruijn, N. G. Reflections on Automath. In *Selected papers on Automath*, Nederpelt, Geuvers, and de Vrijer, Eds. North-Holland, 1994, pp. 201–228.

[11] Dolzmann, A., and Sturm, T. *REDLOG User Manual. Edition 1.0 for Version 1.0.* Universitaet Passau, 1996.

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

[13] Homann, K. *Symbolisches Loesen mathematischer Probleme durch Kooperation algorithmischer und deduktiver Systeme (in German).* PhD thesis, Universitaet Karlsruhe, 1996.

[14] Horn, C. *The Oyster Proof Development System*, 1988.

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

[16] Kutzler, B., and Stifter, S. On the Application of Buchberger's Algorithm to Automated Geometry Theorem Proving. *Journal of Symbolic Computation 2*, 4 (1986), 389–397.

[17] Paulson, L. C. *ML for the Working Programmer.* Cambridge University Press, 1991.

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

[19] Sturmfels, B. *Algorithms in Invariant Theory.* Springer, Vienna, 1993.

[20] van Harmelen, F., Ireland, A., Negrete, S., Smaill, A., and Stevens, A. *The Clam proof planner*, 1989.

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

[22] Wu, W. *Mechanical Theorem Proving in Geometries.* Springer, Vienna, 1994. Translation from chinese.