■ 1: Introduction

We will first sum up a survey paper of the *Theorema* project, giving an overview on the main new ideas of the *Theorema* approach of automated theorem proving included in an existing software system for "doing Mathematics", as *Mathematica*, with references to the publications giving details about these concepts. The current status of the *Theorema* system, so far implemented in *Mathematica 3.0* by Bruno Buchberger, Tudor Jebelean, Franz Kriftner, Mircea Marin, Eleana Tomuta, Daniela Vasaru and the author, is described next, explaining how the current work fits into the project structure. Some important features of the language are described, especially the main difference to ‘usual’ languages of higher order logic: sequence variables. We will further explain why at the current stage of the project we concentrate on one special functor−defined domain: the polynomials.

■ 1.1: The *Theorema* Project

The goal of combining computer algebra and logic was formulated already in [Buchberger, 1985]. The paper [Buchberger, 1993] sketches the ability of computer algebra systems to support theorem proving, at the example of the *Mathematica* system. Structure and goals of the *Theorema* project were specified explicitly in [Buchberger 1996a] for the first time. There some first proofs generated automatically by *Theorema* provers are presented.

In the following, we will stick closely to [Buchberger *et al.*, 1997], where an overview of the current state of the system is given. We collect and explain the basic ideas described in this survey:

■ 1.1.1: A Uniform Working Environment

The main goal of the project is to provide a mathematician’s working environment in which the three basic types of mathematical work are all supported:

− Proving: Deciding whether a provided predicate (an expression with free variables) holds for all variable assignments.

− Solving: Finding variable assignments such that a provided predicate holds.

− Computing: Evaluating expressions (terms, formulae, programs).

One reason for having support of these different tasks combined in one system is that they are dependent of each other:

− Proving an existentially quantified formula is essentially solving.

− Solving and proving both need computing, for example to be able to evaluate functions, to simplify terms.

− During solving, proving can be needed to test whether a value is really a solution or to show that there are no solutions.

− Computing the value of an existentially quantified formula needs finding the ‘existing’ value (and hence solving) or proving the negation of the formula, which is universally quantified.

− During solving, proving can be needed to test whether a value is really a solution or to show that there are no solutions.

When we want to switch between the different working methods by the above correspondences, the best way is to provide a common platform in order to have a uniform language and to preserve the knowledge base and the logic (calculus).

■ 1.1.2: The Problem Solving Cycle

Mathematicians' problem solving cycle consists of the phases of *specifying* a problem, *exploring* it, *proving* conjectures about the new defined concepts, *programming* new knowledge into algorithms and *documenting* the work. Changing the system for different phases in this loop normally means having to change to a different syntax, to a different logic system (e.g. consider the semantics of a C programming environment versus the calculus of a resolution prover). It also means that the knowledge base has to be transferred into the new system, which can also be a source of errors. Imagine for example the verification of a C program with an external prover – even transforming the C code into prover input can be very hard. A system that provides proving, solving and computing support will enable mathematicians to go through this cycle without having to change the system.

– *Specification* means compiling a knowledge base and adding new definitions. In *Theorema*, this collected information can be accessed easily in the other phases, because of the same language and semantics.

– *Exploring* is mainly done using the computing facilities to look at examples and examine available algorithms.

– *Proving* is simplified by letting the system prover do the easy but cumbersome part of the proofs, such that the mathematician can concentrate on the creative steps (finding important lemmata, finding necessary instances for existentially quantified variables, etc.).

– *Programming* is supported by the way certain terms, 'algorithmic' formulae, can be interpreted in the *Theorema* environment: both as logic propositions and as executable programs, whose semantics exactly correspond to the one of the logic formulae. By this, the specification of a program *is* already the program.

– *Documenting*, writing descriptions of the problem solving status, is simplified again by the common platform connecting the different system parts. The current *Mathematica* implementation has the additional property that descriptive text, executable *Theorema* code and automatically generated proofs can be merged into an interactive notebook, where the reader can immediately try out his own test examples using the presented algorithms, or he can change definitions, propositions and try to re–calculate proofs in order to see the behaviour under a modified knowledge base.

■ 1.1.3: A Suitable Software System

In contrast to most of the other systems trying to combine the computational and the proving aspect (like e.g Coq [Huet *et al.*, 1994], Isabelle [Paulson, 1996] or Nuprl [Constable *et al.*, 1986]), our approach is not to expand an existing ATP system by computational facilities. Our approach is based on an established computer algebra system – *Mathematica (*[Wolfram, 1996]). However, we do not expand the whole existing system by a proving package, as it is done in the Analytica system ([Clarke, Zhao, 1993]), but we only use the well defined core language of the system, the *Mathematica* rewrite rule language, that represents a version of higher order equational logic, as explained in [Buchberger, 1996b]. Only at this basic level we have the wanted correspondence between the languages used for proving and for computations. Any of the algorithms provided in *Mathematica*'s huge library has to be derived and verified in their *Theorema* language version, via the *Theorema* provers, before we use it in the system. This assures that we can use such an algorithm in the prover, still being able to state the correctness of the proof.

As it is also done for the Coq prover as part of the CtCoq Proof Environment ([Bertot *et al.*, 1996]), we also want to present proofs in natural language style. The main difference to the mentioned approach is, that we do not only formulate the used inference rules (called 'tactics' in Coq) in natural language, but we try to reach as closely as possible the forms proofs are presented in mathematical textbooks and papers, under different context. *Mathematica* provides us with this possibility to show the same proof in different ways, dependent of the reader's interest:

– In a situation like building up a theory by some definitions and many basic theorems, as in a scientific paper, the reader might only be interested in the overall structure of the proof. Here it can be sufficient to state whether the proof works by simplification, induction, by using only the inference rules of predicate logic … and maybe to list the needed lemmata, such that the reader sees the dependences between the propositions.

– A totally different situation would be to present a proof in interactive math teaching. There the reader must have access to each step of the proof, being able to reconstruct the reasoning in all detail, such that he can learn the applied inference rules as a list of 'proof methods', which he can further use by himself.

In both contexts it would be an advantage not to have a fixed level of detail: The reader of the scientific paper might want to check at least to check some part of a proof in more datail, and the math student would like to hide all the subproofs that are already 'easy' for him, to get better overview on the remaining proof structure.

We achieve this flexibility by generating and presenting natural language proofs as *Mathematica* notebooks. The *Theorema* proof postprocessor takes the internal proof representation (which will later on be of a standardized 'proof term' format), and generates the notebook, by adding all the explanatory text and composing subproofs as cell groups that can be open (readable) or closed (hidden). When a subproof cell group is opened, the first layer gives only the name of the main proof method. Opening an additional layer then shows its actual application.

One of the important creative steps in a mathematician's work is to find a good notation, which reflects the semantics of the underlying notions. The *Mathematica 3.0* user interface is currently the only system where one can define many kinds of new two–dimensional notations, even for executable formulae. One can also change the semantics of given notations or change the syntax, keeping the semantics. In fact, large terms and formulae displayed in one–dimensional notation are very difficult to read and understand, even if not only prefix notation is used. Imagine a linear algebra textbook talking about matrices as lists of lists … This example shows that good typesetting is not only a convenience, but without it displaying the proofs would become basically useless. Having unreadable prover output comes close to having a black box prover that only answers "proved" or "failed".

This way of presenting proofs in textbook style does of course not only rely on finding the right level of detail, good explanatory phrases and a descriptive syntax. It also needs that in the automatic prover we use the same powerful set of inference rules as mathematicians do. This 'proving by natural deduction' means that we cannot stay with a limited set of such rules (or even only one rule, as in resolution) that provide nice theoretical properties. At least for the system's higher order predicate logic prover ([Buchberger, Jebelean, 1997]) it is planned to prove that the 'natural deduction' inference rules can be built by composing elements of a small basic rule set, in order to inherit its properties.

Other inference rules, such as case distinction or induction schemes, are derived by the definition of the corresponding domain by its predicates, constants and functions (most important: the element predicate, describing which elements are in the domain's carrier set). *Mathematica* provides a language construct suitable to build such domain definitions, the module, which in the *Theorema* language is renamed to 'functor', since its functionality is to map function symbols to the domain's function. To do calculations or proofs with terms or formulae that contain objects defined in a functor, we again make use of the extended typesetting facilities of the new version 3.0: An object `obj` defined in a functor `F` can be written as $\underset{\mathtt{F}}{\mathtt{obj}}$, using an underscript.

■ 1.2: Structure of the *Theorema* System

Based on a software system for symbolic computation as implementation environment and language frame, we use a version of higher order logic and the functor concept to build up towers of mathematical domains. To simultaneously create the corresponding theories, we build up a tower of provers in parallel, 'domain provers', that make use of special domain properties and inference rules.

Base of this tower is the general 'natural style' predicate logic prover (designed using the inference rules described e.g. in [Buchberger, Lichtenberger, 1981], but also using the 'less natural' resolution as additional method). Since we want to split up proving tasks as much as possible, equational reasoning is not part of this predicate logic prover, but is treated within a special simplification prover. In the same way we treat proofs needing case distinction not by the predicate logic prover but by a separated case prover. First examples of special provers are an induction prover for natural numbers, an induction prover for lists (both already working) and an induction prover for polynomials (existing as prototype). The special inference rules used there – the induction principles – are derived from the functor definition of natural numbers, lists and polynomials, respectively.

The project members are responsible for the following parts of the system: syntax of the mathematical language (F. Kriftner), generation of knowledge bases from functors (E. Tomuta), predicate logic prover (T. Jebelean), simplification prover (M. Marin), inductive prover for natural numbers (B. Buchberger), inductive prover for lists (D. Vasaru), inductive prover for polynomials (the author).

Documentations about these parts can be found in [Buchberger, Kriftner, 1997], [Tomuta, 1997], [Buchberger, Jebelean, 1997], [Buchberger, Marin, 1997], [Buchberger, 1997], [Buchberger, Vasaru, 1997], the authors work is described here for the first time.

The core of the proof post–processor, generating the natural language output, is primarily only able to treat predicate logic proofs, but this part of the system grows with the number of implemented provers: Each of the above special provers is accompanied by a corresponding extension of the post–processor, such that it is enabled to formulate the new inference rules in natural language.

Future extensions of the system will also make efficient 'black box' algorithms available, like provers for special domains or special solvers, such that the user does not have to leave his working environment.

Other future plans are a 'prover generator', taking a functor and delivering a special prover for domains created with this functor, and a 'proof analyzer', that takes a failed proof and proposes lemmata that would enable the proof. This analyzer could even be part of a 'theory generator', that proves a proposition by building and proving the tree of necessary lemmata.

■ 1.3: Language Overview

We only want to present the main features of the *Theorema* language by an example, such that the reader can understand our notation. Details can be found in [Buchberger, Kriftner, 1997]. For this, we examine the formula

$$\underset{q}{\forall}\ \langle\overline{p_0}\rangle * q = q * \langle\overline{p_0}\rangle\ ,$$

that will later on occur similarly during one of our proofs.

■ 1.3.1: Basic Elements

Smallest building blocks of the language are *object symbols*, like $\ast$, q, $\overline{p_0}$, $\langle\,\rangle$. They can be used as constants or (bound or free) variables. In the present implementation, we have to be careful not to use any symbols that are interpreted by the *Mathematica* kernel or even already the *Mathematica* frontend, but we plan to suppress this evaluation more rigorously in order to have access to all ‘standard’ mathematical symbols, like + or = .

Terms are composed using object symbols or terms, by *function application*: basically, this is written using prefix notation with square brackets, like $\ast[\langle\overline{p_0}\rangle, q]$, but we also allow infix (as used above), pre–postfix (as for $\langle\ldots\rangle$) and postfix notation. Special ‘sequence’ symbols (explained later), marked by an overbar, are only allowed to appear as argument to a function.

Special function symbols are the $=$, denoting equality, and the angle brackets $\langle\ldots\rangle$, used to construct tuples. In deviation to the cited language paper, we will use the normal equality symbol, not the congruence $\equiv$, to stay closer to standard mathematical notation.

Formulae are composed by *quantifying* a term, making some of the appearing object symbols into bound variables, and by using propositional connectives. Furthermore the ‘$\forall$’–quantifier can be left out, so formulas with free variables are interpreted as universally quantified. Out of its actual proof context, the above formula would have such a free variable: $\overline{p_0}$. During the proof, however, $\overline{p_0}$ has been declared to be a constant before using the formula.

■ 1.3.2: Readability

Our language allows some abbreviations, by using typed variables and functor/domain declarations. Initially, the above formula may look like this:

$$\underset{q\in \mathrm{Poly}[M]}{\forall}\ \mathrm{Poly}[M][\ast][\langle\overline{p_0}\rangle, q] = \mathrm{Poly}[M][\ast][q, \langle\overline{p_0}\rangle]\ .$$

An additional way of displaying function application is used for functors. In our example Poly is a functor creating a polynomial domain from an input monomial domain, M . We use underscripts and get

$$\underset{q\in \mathrm{Poly}[M]}{\forall}\ \langle\overline{p_0}\rangle \underset{\mathrm{Poly}[M]}{\ast} q = q \underset{\mathrm{Poly}[M]}{\ast} \langle\overline{p_0}\rangle\ .$$

This formula is now universally quantified over M . In an environment where M is declared to be a constant, we can further improve readablilty by setting a new domain constant (here P) to the value Poly[M] . For this, we state "Let M be arbitrary but fixed. Let P be Poly[M] .", which allows us to use

$$\underset{q\in P}{\forall}\ \langle\overline{p_0}\rangle \underset{P}{\ast} q = q \underset{P}{\ast} \langle\overline{p_0}\rangle\ .$$

Furthermore (if it supports readability) we can declare $\ast$ to be by default in the signature of P, "In the following, we omit the subscript of $\underset{P}{\ast}$.", yielding

$$\underset{q\in P}{\forall}\ \langle\overline{p_0}\rangle \ast q = q \ast \langle\overline{p_0}\rangle\ .$$

Finally the type of q can be declared, by stating "Let q range over P .". Now we can omit the type information below the quantifier and the formula looks like our initial example.

All these declarations often take place at the beginning of a document or of a proof, but they can also be only valid for e.g. just one chapter of a paper. Therefore the scope of the declarations has to be stated if it is not obvious.

■ 1.3.3: Sequence Variables and Induction

A certain category of symbols, called *sequence symbols*, has special properties. They are denoted using an overbar. Their semantics justify the name: Sequence constants denote finite ordered sequences of objects, sequence variables can be replaced by a finite sequence of objects during a substitution. The sequence variables' semantics is already available in *Mathematica*, where they are written with three underscores instead the overbar. The concept of sequence symbols gives rise to one important additional inference rule, available in all provers, the 'sequence induction':

$$\frac{A_{\bar{x}\leftarrow},\ \underset{x,\bar{x}}{\forall} A \Longrightarrow A_{\bar{x}\leftarrow x,\bar{x}}}{\underset{\bar{x}}{\forall} A}$$

This is applied in the following way, merged with the steps of how to prove the implication:

Let $\bar{x}$ be free in A. To prove $\underset{\bar{x}}{\forall} A$, we prove the base case of the sequence induction, the formula that results by replacing $\bar{x}$ by nothing (= the empty sequence). Then we prove the induction step. For this we take new (= not appearing in A) constants $\dot{x}_0$ and $\overline{x_0}$ arbitrary but fixed and assume the induction hypothesis (given by replacing $\bar{x}$ with $\overline{x_0}$ in A), then we prove the induction conclusion, generated by replacing $\bar{x}$ with the sequence $\dot{x}_0,\ \overline{x_0}$ in A.

■ 1.4: Polynomials in *Theorema*

■ 1.4.1: Reasons for looking at Polynomials

Introducing multivariate polynomials as next functor/prover pair is motivated by the current 'benchmark' project goal to show an example of a paper presenting Gröbner bases, where the main theorem connects the non–computational with the algorithmic part of the theory, see e.g. [Buchberger, Kriftner, 1997]. For this, we have to build a theory of polynomials first.

From the system development's point of view, there are three main reasons why we concentrate on building an inductive prover for polynomials in *Theorema*. This prover extends the sequence of available special domain provers in the following ways:

– The polynomial prover is a first example of a prover derived from a functor that is not at the base of the functor tower. The natural numbers functor is basic in this sense – the prover for natural numbers is derived from a functor without argument domain, the representation of the natural numbers is 'ground' in the sense that it does not depend on the representation of elements in another domain. The list prover is already derived from a functor with domain argument, namely the domain the list elements come from. But the operations defined for lists (as prepending, concatenation, length, …) are defined independently of operations in the argument domain. Our polynomial functor is now the first instance of a parametric functor (= having argument domains), where the definitions of the objects actually make use of the argument domain objects, here the monomial predicates, constants and functions. An example is the polynomial constant one, which is defined by a term containing the monomial constant one. These dependencies become visible in the polynomial prover by the fact that the prover has to access the monomial knowledge base. These properties of the monomials are in fact the only interface to them, the polynomial proofs stay independent of the actual presentation of monomials.

– The polynomial functor we use introduces two new language elements: a case distinction and a 'where' construct. The case distinction could in fact be represented by the already available propositional connectives, but since we want to treat case distinction during proofs not by the predicate logic prover but by a special prover, we use this new construct, similarly to how it is used in the working language of mathematicians. The case distinction has special semantics, resulting in efficient abbreviations in comparison with its representation using only and, or, implication, … The used 'where' quantifier is useful to abbreviate terms that have several equal subterms. It is applied to assign a value to a free variable in a term. By this, the equal subterms can be replaced by a new free variable, then the overall term is quantified and the variable becomes bound to the subterm.

– The polynomial prover uses induction principles in a different way. As it was described at the semantics of sequence variables, there is a basic principle of sequence induction available throughout the system. The current natural numbers prover does not use it, but introduces its own induction principle as inference rule, which looks like this:

$$\frac{A_{n\leftarrow 0},\ \underset{n}{\forall} A \Longrightarrow A_{n\leftarrow n^+}}{\underset{n}{\forall} A}\ .$$

This new inference rule is derived from the element predicate ϵ of the natural numbers functor. There it is defined like this:

$$\begin{array}{ll} (0\in N) & 0\in N; \\ (\in N) & n^+ \in N :\Longleftrightarrow n \in N; \end{array}$$

(The functor syntax is described in detail later, but obviously this is a recursive definition of the element predicate, using the constant 0 and the successor function $^+$, written as superscript.)

Of course it would be possible to define another natural numbers functor (corresponding to another representation of $\mathbb{N}$), e.g.

$$\begin{array}{ll} (\langle\rangle \in NSeq) & \langle\rangle \in NSeq; \\ (\in NSeq) & \langle 1, \bar{n}\rangle \in N :\Longleftrightarrow \langle\bar{n}\rangle \in N; \end{array}$$

based on sequences (a natural number n is an n–tuple with elements 1) , where the initial sequence induction could be used for proving. Since we do not want to restrict the user to certain representations, we support both kinds of definitions. The same alternatives are available when we want to define lists (over the element domain E): One could use the constant Nil and the function Cons and would get the induction principle

$$\frac{A_{x\leftarrow Nil},\ \underset{e\in E,x}{\forall} A \Longrightarrow A_{x\leftarrow Cons[e,x]}}{\underset{x}{\forall} A}\ ,$$

or we define lists via sequence variables. The available list induction prover is based on a list functor using sequence variables, therefore it only uses sequence induction.

The main difference of our polynomial functor (over the monomial domain M) to the former two is, that from the element predicate

$$\begin{array}{ll} (\langle\rangle \in P) & \langle\rangle \in P; \\ (\langle m\rangle \in P) & \langle m\rangle \in P :\Longleftrightarrow m \in M; \\ (\in P) & \langle m, n, \bar{p}\rangle \in P :\Longleftrightarrow m \in M \wedge \langle n, \bar{p}\rangle \in P \wedge m \underset{M}{>} n; \end{array}$$

a specific polynomial induction principle

$$\frac{A_{\bar{p}\leftarrow},\ \underset{m\in M}{\forall} A_{\bar{p}\leftarrow m},\ \underset{m\in M,\ \langle n,\bar{p}\rangle \in P}{\forall} A_{\bar{p}\leftarrow n,\bar{p}} \wedge m \underset{M}{>} n \Longrightarrow A_{\bar{p}\leftarrow m,n,\bar{p}}}{\underset{\bar{p}}{\forall} A}$$

can be derived and used, but sequence induction can also be used. This influences the design of the prover, since it can first try (the simpler) sequence induction, but also has the more powerful polynomial induction principle available.

■ 1.4.2: Overview of the Working State

Our starting point was the definition of the polynomial functor, as it is shown in the chapter 'Polynomial Theory'. Our first attempt was to build an ad–hoc version of a polynomial prover similarly to Vasaru's list prover. This prototype does not yet care about type checking and only implements an algorithm of using polynomial induction recursively and calling auxiliary available provers for the subgoals. Furthermore it only proves universally quantified equations and does not apply sequence induction. It was used to try first proofs, to test the formatting of the output notebook and to test how to access the auxiliary provers.

Soon it became visible that the automatically produced proof notebooks did not correspond to our notion of 'natural' proofs. In order to clarify this notion, we proceeded in two steps: We examined the output notebooks in detail and marked the 'unnatural' proof parts, and we manually created proof versions in notebook format to be able to compare the versions, and to have models of proofs as goals that can be aimed at by future versions of the prover.