Theorema: An Integrated System for Computation and Deduction in Natural Style

The *Theorema* Group*
(B. Buchberger, K. Aigner, C. Dupré, T. Jebelean, F. Kriftner, M. Marin, K. Nakagawa, O. Podişor, E. Tomuţa, Y. Usenko, D. Văsaru, W. Windsteiger)

Research Institute for Symbolic Computation, A-4232 Hagenberg, Austria
`buchberger@risc.uni-linz.ac.at`,
`http://www.risc.uni-linz.ac.at`

Abstract. The *Theorema* project aims at integrating computation and deduction in a system that can be used by the working scientist for building and checking mathematical models, including the design and verification of new algorithms. Currently, the system uses the rewrite engine of the computer algebra system *Mathematica* for building and combining a number of automatic/interactive provers (high-order predicate-logic, induction for lists/tuples and natural numbers, etc.) in natural deduction style and in natural language presentation. These provers can be used for defining and proving properties of mathematical models and algorithms, while a specially provided "computing engine" can execute directly the logical description of these algorithms.

1 Introduction

The aim of the *Theorema* project is to provide an easy-to-use intelligent environment for the working scientist, which integrates the computating capabilities of a computer algebra system with the deduction capabilities of automatic theorem provers. This system can be used interactively for bulding and checking mathematical models (which also means designing and verifying new algorithms) as well as for writing (interactive) textbooks in mathematics, computer science, and engineering, and additionally for the training of students in mathematical logic and computer science.

The system is currently under development at RISC-Linz. In a first phase of the development cycle a number of provers have been written using the rewrite language of the computer algebra system *Mathematica* [1, 2, 8]. We are now in a second phase of the development, in which these provers are integrated with each-other and with a common user interface, everything being implemented in *Mathematica* packages using a common style. Some components are already working: the user interface, the logical evaluator, the common mechanism for integrating provers, and the provers for high-order predicate-logic prover, for

* Partially supported by the Austrian Science Foundation (FWF), project FO-1302 (SFB).

Presented at CADE'98 (International Conference on Computer Aided Deduction), Lindau, Germany – Workshop on Integratinng Proving and Computing. July 1998.

equational simplification, for induction on lists and on natural numbers. A special prover for sets, one for mathematical analysis, and one for functor-built domains are under development.

From the many powerful deduction systems referenced in the literature, probably the most relevant to our approach are Analytica [4] and Redlog [5], which add proving support to an existing computer algebra system. The distinctive features of *Theorema* are listed below and detailed in the subsequent sections.

Natural language presentation using *Mathematica* notebooks. The proofs produced by our system are presented in natural (English) language, similar to the one which is used in scientific textbooks (support for various languages can be added easily). Both for input and for output, the description of mathematical models and of the algorithms uses the language of high-order predicate logic – the formulae being rendered in the usual two-dimensional mathematical notation.

Natural style deduction and integration of provers. The inference mechanisms used by the various provers are modeled upon the proving style of working mathematicians. Moreover, combining these provers is also done in the same style: depending on the domain to which the formulae refer, a prover incorporating specific knowledge about this domain will be called. *Theorema* uses a general mechanism for integrating several domain–specific provers according to a specified strategy. The end–user accesses several high-level provers defined in this way.

Combining proving and computing. The provers can be used for defining and proving properties of mathematical models and algorithms, while a specially provided "computing engine" can execute the logical description of these algorithms directly. This requires a careful separation of the "object–level" and the "meta–level" of the language used.

2 The User Interface

We want to come as close as possible to the language used traditionally by scientists. In *Theorema*, the description of mathematical models and of the algorithms uses the language of higher–order predicate logic – the formulae being rendered in the usual two-dimensional mathematical notation.

The user can define new theories and properties using declarations like `Theory`, `Definition`, `Lemma`, `Theorem` etc. and then he can call the provers by `Prove` and the evaluation engine by `Compute`, indicating the knowledge base `KB` and the prover to be used – section 3 presents a sample session.

The proofs produced by our system are presented in natural (English) language, similar to the one which is used in scientific textbooks. An example of such a proof is presented in Fig.1. The brackets on the right-hand side represent the nested structure of the cells, which corresponds to the structure of the proof. The user can open or close the cells by mouse click, thus being able to study the proof at the desired level of detail. (In the figure, italic comments show which cells are closed.)

The external (notebook) representation of the proof is produced from an internal representation (the proof object - see section 4) by a set of display packages, thus is an activity separated from the proof search. First a simple analyser generates the structure of the cells from the tree structure of the proof object, then the texts are displayed at the appropriate places. Support for various languages can be easily added by modifying the specific packages which display these texts.

3 Interfacing Proving, Computing, and Programming

The *Theorema* system integrates *provers* of logical formulae with *computing*, i. e. interpreting the same logical formulae (whenever possible). Thus, the language in which the problems are specified is the same as the language in which the algorithms are programmed: this is a novel approach which closes completely the gap between proving and computing. However, in this situation special care has to be taken: formulae which are intented to be treated by the provers should not be evaluated. In fact, since the user will need to experiment with various versions of an algorithm (based on various sets of formulae), even formulae intented for evaluation should not be computed immediately, but only in certain contexts. Moreover, in the *Theorema* project we use the same language (Mathematica 3.0) as the *working language* for the user (which he needs in order to organize the knowledge he defines and to call various functions of the system) as well as the *programming language* for implementing the system's provers, evaluators, etc.

In order to cope with these problems, when an expression is input to the system, *Theorema* clearly distinguishes between the meta-level (where the expresions should be evaluated) and the object-level (formula-level – where the expressions are kept unchanged). This is done by wrapping the object-level expressions into special constructs like `Proposition[...]`, `Theory[...]`, etc. (called *environments*) in the user language, and others like `•lf[...]`, `•[...]` in the language provided for the system programmers. These constructs are treated before any evaluation takes place (by extending some basic functions of the *Mathematica* front-end) and transform the object-level expressions by replacing each interesting function name by a new one which has no meaning for the built-in *Mathematica* evaluator. This translation is done using a translation table and a special prefix character (forbidden to the user) and allows one-to-one back translation for the possible evaluation. The following dialog shows how, in the *Theorema* standard session, no evaluations happen inside a formula, unless "Compute" is used explicitely:

Input: $\bullet lf[\text{“1”}, 1 < 0 \Leftrightarrow X > 0]$
Output: $\bullet lf[\text{“1”}, 1 < 0 \Leftrightarrow X > 0]$
Input: Compute$[\bullet lf[\text{“1”}, 1 < 0 \Leftrightarrow X > 0]]$
Output: $X > 0 \Rightarrow False$

The object-level formulae created by the user are organized in *knowledge-bases* by means of simple and intuitive constructs (hierarchical organization is also possible) provided by the *Theorema user language*. For instance, one can type:

Input: $\text{Definition}[\text{“divides”}, \text{any}[n, m],$
$\quad \text{div}[n, m] \Leftrightarrow (\exists_{q \in N}(m = nq))\ \text{“non-algorithmic”}]$

Input: $\text{Proposition}[\text{”}divides\text{”}, \text{any}[n, m],$
$\quad \text{div}[n, m] \Leftrightarrow (\exists_{q=1,\ldots,m}(m = nq))\ \text{“algorithmic”}]$

The any[n,m] is used here instead of a universal quantifier in the formula, this is useful when in the same environment several formulae use the same variables. From now on we can refer to the formula(e) given in the environment by `Definition["divides"]`, or `Proposition["divides"]`, respectively.

Both computing and proving are always performed relatively to a certain knowledge base, which is specified as the argument of the prover/evaluator call or as a previously defined default. Thus, the session above may continue with:

Input: $\text{Compute}[\text{div}[3, 6], KB \to \text{Definition}[\text{“divides”}]]$
Output: $\exists_{q \in N}(6 == 3q)$
Input: $\text{Prove}[\text{Proposition}[\text{“divides”}],$
$\quad KB \to \text{Definition}[\text{“divides”}], Prover \to NatNumProver]$
Output: $proof - notebook$
Input: $\text{Compute}[\text{div}[4, 999], KB \to \text{Proposition}[\text{“divides”}]]$
Output: $False$

The *Theorema* system provides the interpretation of all (computable) logic constructs, as well as the elementary functions of basic mathematical domains (sets, natural numbers, lists), which can be used at user request. For the situations when the user is mainly interested in computing, we provide a *Theorema computational session*, in which every input given to the system is evaluated automatically just as if it was wrapped in a call to “Compute”.

Input: ComputationalSession[“Division”];
Input: $\text{div}[7, 49]$
Output: $True$
Input: EndComputationalSession[ ]

4 Combining Different Provers

Interesting mathematical constructions usually include knowledge from various mathematical domains (e.g. sets, natural numbers, lists/tuples, mathematical analysis, etc.). In our system, for each such basic domain we construct a special *basic prover*, which integrates knowledge specific to that domain, and additionally a higher-order predicate logic prover to handle the “purely logic” steps. In order to integrate these provers, *Theorema* uses a general proof-search mechanism (see [7]), which combines the specific deduction steps in order to realize a higher level *user prover*. For instance, the user prover `NaturalInductionProver` used for producing the proof in Fig.1 combines the basic provers: predicate logic

(`PredLog`), natural numbers specific induction (`NNInd`), and equational proving by simplification (`Simp`).

The proof search engine uses an internal tree-like data structure - *the proof object*, which is also the internal representation of the proof. Consequently, as in [6], the derivation trees in *Theorema* contain "OR"-nodes corresponding to the various (disjunctive) alternatives of solving a proof situation and "AND"-nodes corresponding to the (conjunctive) composition of subproofs. In addition, we also use "IF"-nodes for the cases in which a proof situation is reduced differently depending on whether a certain subproof is successful or not.

The proof search engine combines various separate derivation trees obtained by calling individual provers, into the global proof tree. During proof search, the proof values of the nodes in the derivation tree are computed bottom up, based on its recursive structure. The choice of the basic provers to be applied to the current proof situation is based on a strategy which is specific to the corresponding user prover. This strategy is specified by a list of simple clauses, each indicating a condition (on the proof situation) and the basic prover to apply in this case. For instance, the **NaturalInductionProver** specifies:

```
SetStrategy[Clause[GroundEquality,    Simp],
            Clause[UniversalEquality, NNInd],
            Clause[True,              PredLog]];
```

This means that if the goal is a ground equality, then the simplification prover is applied, if it is an universal equality then the induction prover is applied, otherwise the predicate logic is applied. Fig.1 shows a proof produced by this user prover, in which simplification steps alternate with induction steps.

Interaction. Since one cannot hope to solve automatically all the proof situations which occur in non-trivial models, provinding appropriate interaction methods is crucial for the system. A first iteraction level is provided by the user session (see Section 3), which allows defining and refining of the models according to the results of proving and computations.

However, the system allows to interact with the provers themselves, more specifically we design interactive user-level provers using the same proof control control mechanism, (enhanced with specific interaction packages) which combines the same basic provers. Fig.2 shows a snapshot of the screen when using an interactive version of the predicate logic prover (this was designed for instruction purposes - see [3]). Various windows display information about the status of the proof (including the proof-notebook, which contains still unsolved proof situations shown in gray). The button-palette in the up-right corner alows the interactive control: the user may unfold the proof step by step, remove some unnecessary assumptions in order to reduce the search space, give hints in form of intermediate goals to be proven, navigate through the unfinished proof and choose additional proving strategies, etc.

Prove:

(0+) $\forall_{m} (0 + m = m)$,

under the assumptions:

(+0) $\forall_{m} (m + 0 = m)$,

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

We prove (0+) by induction on m.

Induction Base:

(0+.IB) $0 + 0 = 0$.

A proof by simplification of (0+.IB) works. *(closed cell)*

Induction Hypothesis:

(0+.IH) $0 + m_1 = m_1$

Induction Step:

(0+.IS) $0 + (m_1)^{+} = (m_1)^{+}$.

A proof by simplification of (0+.IS) works.

Simplification of the lhs term:

$0 + (m_1)^{+} =$ by (++)

$(0 + m_1)^{+} =$ by (0+.IH)

$(m_1)^{+} \rfloor$

Simplification of the rhs term:

$(m_1)^{+} \rfloor$

Fig. 1. Proof notebook produced by the NaturalInductionProver.

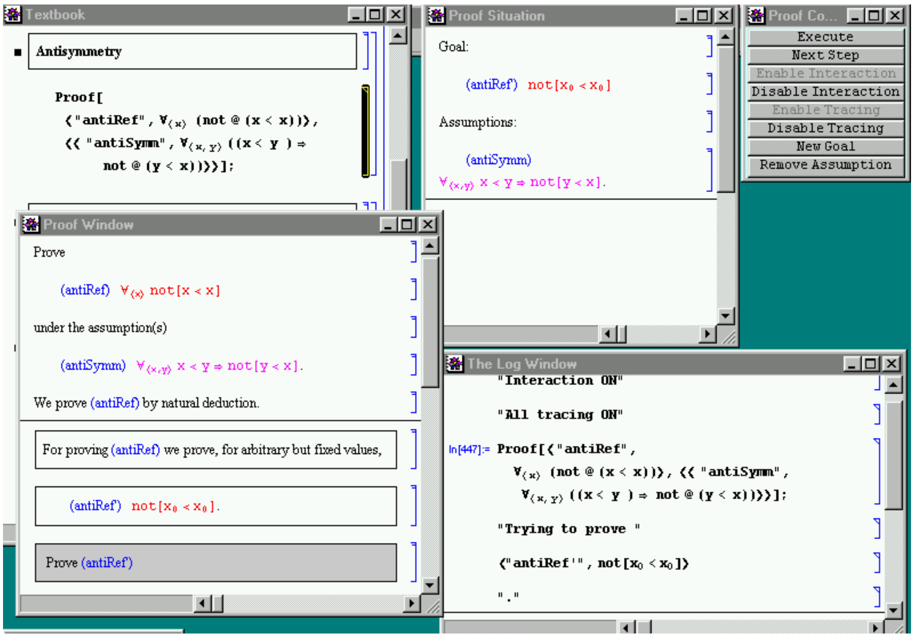

Fig. 2. Screenshot of the interactive prover.

References

1. B. Buchberger: Using Mathematica for Doing Simple Mathematical Proofs. Invited paper, in: Proceedings of the 4th Mathematica Users' Conference, Tokyo, 1996, Wolfram Media Publishing, pp. 80–96.
2. B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuta, D. Vasaru: A Survey of the *Theorema* Project, Proceedings of ISSAC'97, W. Kuechlin (ed.), ACM Press 1997, pp. 384–391.
3. B. Buchberger, T. Jebelean, D. Vasaru: Theorema: A System for Formal Scientific Training in Natural Language Presentation, ED-MEDIA'98, Springer LNCS, in print.
4. E. M. Clarke, X. Zhao: Analytica - A theorem prover for Mathematica. The Mathematica Journal 3(1) 1993, 56-71.
5. A. Dolzmann, T. Sturm: REDLOG User Manual. Edition 1.0 for Version 1.0.Universitaet Passau. 1996
6. N. J. Nillson: Principals of Artificial Intelligence, Tioga Publishing, Palo Alto, 1980.
7. E. Tomuţa: Proof Control in the *Theorema* System. PhD Thesis, RISC-Report 1998, in print.
8. D. Vasaru, T. Ida, B. Buchberger (eds.): First International Theorema Workshop, RISC-Report 97-20, 1997 `http://www.risc.uni-linz.ac.at/library/`