Towards Practical Reflection for Formal Mathematics

—extended abstract—

Martin Giese and Bruno Buchberger

RISC, Johannes Kepler University,
A-4232 Schloß Hagenberg, Austria
{bruno.buchberger|martin.giese}@risc.uni-linz.ac.at

Abstract. We describe a design for a system for mathematical theory exploration that can be extended by implementing new reasoners using the logical input language of the system. Such new reasoners can be applied like the built-in reasoners, and it is possible to reason about them, e.g. proving their soundness, within the system. This is achieved in a practical and attractive way by adding reflection, i.e. a representation mechanism for terms and formulae, to the system's logical language, and some knowledge about these entities to the system's basic reasoners. The approach has been evaluated using a prototypical implementation called Mini-Tma. It will be incorporated into the Theorema system.

Introduction

Mathematical theory exploration consists not only of inventing axioms and proving theorems. Amongst other activities, it also includes the discovery of algorithmic ways of computing solutions to certain problems, and reasoning about such algorithms, e.g. to verify their correctness. What is rarely recognized is that it also includes the discovery and validation of useful techniques for proving theorems within a particular mathematical domain. In some cases, these reasoning techniques might even be algorithmic, making it possible to implement and verify a specialized theorem prover for that domain.

While various systems for automated theorem proving have been constructed over the past years, some of them specially for mathematics, and some of them quite powerful, they essentially treat theorem proving methods as a built-in part of the services supplied by a system, in general allowing users only to state axioms and theorems, and then to construct proofs for the theorems, interactively or automatically. An extension and adaptation of the theorem proving capabilities themselves, to incorporate knowledge about appropriate reasoning techniques in a given domain, is only possible by stepping back from the theorem proving activity, and modifying the theorem proving software itself, programming in whatever language that system happens to be written.

We consider this to be limiting in two respects:

– To perform this task, which should be an integral part of the exploration process, the user needs to switch to a different language and a radically different way of interacting with the system. Usually it will also require an inordinate amount of insight into the architecture of the system.
– The theorem proving procedures programmed in this way cannot be made the object of the mathematical studies inside the system: e.g., there is no simple way to prove the soundness of a newly written reasoner within the system. It's part of the system's code, but it's not available as part of the system's knowledge.

Following a proposal of Buchberger [3, 4], and as part of an ongoing effort to redesign and reimplement the Theorema system [5], we will extend that system's capabilities in such a way that the definition of and the reasoning about new theorem proving methods is possible seamlessly through the same user interface as the more conventional tasks of mathematical theory exploration.

In this paper, we briefly outline our approach as it has been implemented by the first author in a prototype called Mini-Tma, a Mathematica [9] program which does not share any of the code of the current Theorema implementation. Essentially the same approach will be followed in the upcoming new implementation of Theorema.

A more complete account of our work, including a discussion of some foundational issues, and various related work, is given in [6].

The Framework

We start from the logic previously employed in the Theorema system, namely higher-order predicate logic with sequence variables. Sequence variables [8] represent sequences of values, and are written like $\overline{xs}$. Amongst many other uses, sequence variables happen to be convenient in statements about terms and formulae, since term construction in our logic is a variable arity operation.

In order to make reasoning about syntactic structures as attractive as possible, we build a *quotation mechanism* into the logic. For the representation of symbols, we require the signature to contain a *quoted version* of every symbol. Designating quotation by underlining, we write the quoted version of a as $\underline{\mathsf{a}}$, the quoted version of f as $\underline{\mathsf{f}}$, etc. For compound terms, we reuse the function application brackets $[\cdots]$ for term construction. This allows us to write $\underline{\mathsf{f}}[\underline{\mathsf{a}}]$ as quoted form of $f[a]$. To further simplify reading and writing of quoted expressions, Mini-Tma allows underlining a whole sub-expression as a shorthand for recursively underlining all occurring symbols.

For variable binding operators, like quantifiers, we prefer an explicit representation to e.g. higher-order abstract syntax. The only derivation from a straight-forward representation is that we restrict ourselves to λ-abstraction as the only binding operator. Thus $\underset{x}{\forall}\,\mathsf{p}[x]$ is represented as

$$\underline{\mathsf{ForAll}}[\underline{\lambda}[\underline{x}, \underline{\mathsf{p}}[\underline{x}]]]$$

where $\underline{\lambda}$ is an ordinary (quoted) symbol, that does not have any binding properties.

The logical input language can be used to write programs. The standard computation mechanism interprets conditional equations as conditional rewriting rules. While this is the basic computation mechanism, we shall see later on that it is possible to define new computation mechanisms in Mini-Tma.

Mini-Tma does not include a predefined concept of *proving mechanism*. Theorem provers are simply realized as computation mechanisms that simplify a formula to True if they can prove it, and return it unchanged (or maybe partially simplified) otherwise.

Defining Reasoners

Since reasoners are just special computation mechanisms in Mini-Tma, we are interested in how to add a new computation mechanism to the system. This is done in two steps: first, using some existing computation mechanism, we define a function that takes a (quoted) term and a set of (quoted) axioms, and returns another (quoted) term. Then we tell the system that the defined function should be usable as computation mechanism with a certain name.

For instance, to define a reasoner that shifts parentheses in sums to the right, e.g. transforming the term Plus[Plus[a, b], Plus[c, d]] to Plus[a, Plus[b, Plus[c, d]]], we start by defining a function that will transform *representations* of terms, i.e. $\underline{\mathsf{Plus}}[\underline{\mathsf{Plus}}[\underline{\mathsf{a}}, \underline{\mathsf{b}}], \underline{\mathsf{Plus}}[\underline{\mathsf{c}}, \underline{\mathsf{d}}]]$ to $\underline{\mathsf{Plus}}[\underline{\mathsf{a}}, \underline{\mathsf{Plus}}[\underline{\mathsf{b}}, \underline{\mathsf{Plus}}[\underline{\mathsf{c}}, \underline{\mathsf{d}}]]]$. We can do this with the following definition:

$$
\begin{array}{l}
\text{Axioms}\,[\text{"shift parens"}, \text{any}[s, t, t_1, t_2, acc, l, ax, comp], \\
\quad \mathsf{simp}[t, ax, comp] = \mathsf{add\text{-}terms}[\mathsf{collect}[t, \{\}]] \\
\quad \mathsf{collect}\,[\underline{\mathsf{Plus}}[t_1, t_2], acc] = \mathsf{collect}[t_1, \mathsf{collect}[t_2, acc]] \\
\quad \mathsf{is\text{-}symbol}[t] \Rightarrow \mathsf{collect}[t, acc] = \mathsf{cons}[t, acc] \\
\quad \mathsf{head}[t] \neq \underline{\mathsf{Plus}} \Rightarrow \mathsf{collect}[t, acc] = \mathsf{cons}[t, acc] \\
\quad \mathsf{add\text{-}terms}[\{\}] = \underline{0} \\
\quad \mathsf{add\text{-}terms}[\mathsf{cons}[t, \{\}]] = t \\
\quad \mathsf{add\text{-}terms}[\mathsf{cons}[s, \mathsf{cons}[t, l]]] = \underline{\mathsf{Plus}}[s, \mathsf{add\text{-}terms}[\mathsf{cons}[t, l]]] \\
]
\end{array}
$$

The main function is simp, its arguments are the term t, the set of axioms ax, and another computation mechanism *comp*, which may be used to recursively evaluate subterms. Given these axioms, we can ask the system to simplify a term:

$$
\begin{array}{l}
\text{Compute}[\mathsf{simp}[\underline{\mathsf{Plus}}[\underline{\mathsf{Plus}}[\underline{\mathsf{a}}, \underline{\mathsf{b}}], \underline{\mathsf{Plus}}[\underline{\mathsf{c}}, \underline{\mathsf{d}}]]], \{\}, \{\}], \text{by} \rightarrow \text{ConditionalRewriting}, \\
\qquad \text{using} \rightarrow \{\text{Axioms}[\text{"shift parens"}], \ldots\}]
\end{array}
$$

We are passing in dummy arguments for ax and *comp*, since they will be discarded anyway. Mini-Tma will answer with the term $\underline{\mathsf{Plus}}[\underline{\mathsf{a}}, \underline{\mathsf{Plus}}[\underline{\mathsf{b}}, \underline{\mathsf{Plus}}[\underline{\mathsf{c}}, \underline{\mathsf{d}}]]]$.

We can now make simp known to the system as a computation mechanism. After typing

$$\text{DeclareComputer}[\text{ShiftParens}, \mathsf{simp}, \text{by} \rightarrow \text{ConditionalRewriting}, \text{using} \rightarrow \{\text{Axioms}[\text{"shift parens"}, \ldots]\}]$$

the system recognizes a new computation mechanism named ShiftParens. When we ask it to

$$\text{Compute}[\mathsf{Plus}[\mathsf{Plus}[\mathsf{a}, \mathsf{b}], \mathsf{Plus}[\mathsf{c}, \mathsf{d}]], \text{by} \rightarrow \text{ShiftParens}]$$

we will receive the answer $\mathsf{Plus}[\mathsf{a}, \mathsf{Plus}[\mathsf{b}, \mathsf{Plus}[\mathsf{c}, \mathsf{d}]]]$.

Reasoning About Logic

To prove statements about the terms and formulae of the logic, we need a prover that supports structural induction on terms, or *term induction* for short. An interesting aspect is that terms in Mini-Tma, like in Theorema, can have variable arity, and arbitrary terms can appear as the heads of complex terms. Sequence variables are very convenient in dealing with the variable length argument lists. Still, an induction schema needs to be employed that simultaneously performs induction over the depth of terms, and over the lengths of argument lists. Using the mechanism outlined in the previous section, were have implemented a simple term induction prover within Mini-Tma's logical input language.

Reasoning About Reasoners

At this point, it should come as no surprise that it is possible to use Mini-Tma to reason about reasoners written in Mini-Tma. The first application that comes to mind is proving the soundness of new reasoners: they should not be able to prove incorrect statements. Other applications include completeness for a certain class of problems, proving that a simplifier produces output of a certain form, etc.

Our current approach to proving soundness is to prove that anything a new reasoner can prove is true with respect to a model semantics. Or, for a simplifier that simplifies t to t', that t and t' have the same value with respect to the semantics. This approach has also been taken in the ACL2 system [7] and its predecessors.

Similarly to ACL2, we supply a function $\mathsf{eval}[t, \beta]$ that recursively evaluates a term t under some assignment β that provides the meaning of symbols. To prove the soundness of ShiftParens, we have to show

$$\mathsf{eval}[\mathsf{simp}[t, ax, comp], \beta] = \mathsf{eval}[t, \beta]$$

for any term t, any ax and $comp$ and any β with $\beta[\underline{0}] = 0$ and $\beta[\underline{\mathsf{Plus}}] = \mathsf{Plus}$. With a strengthening of the induction hypothesis, and some adaptation of the induction prover, Mini-Tma can prove this statement automatically.

Ultimately, we intend to improve and extend the presented approach, so that it will be possible to successively perform the following tasks within a single framework, using a common logical language and a single interface to the system:

1. define and prove theorems about the concept of Gröbner bases [1],
2. implement an algorithm to compute Gröbner bases,
3. prove that the implementation is correct,
4. implement a new theorem prover for statements in geometry based on coordinatization, and which uses our implementation of the Gröbner bases algorithm,
5. prove soundness of the new theorem prover, using the shown properties of the Gröbner bases algorithm,
6. prove theorems in geometry using the new theorem prover, in the same way as other theorem provers are used in the system.

Though the case studies performed so far are comparatively modest, we are convinced that the outlined approach can be extended to more complex applications.

References

1. Bruno Buchberger. Ein algorithmisches Kriterium für die Lösbarkeit eines algebraischen Gleichungssystems. *Aequationes Math.*, 4:374–383, 1970. English translation published in [2].
2. Bruno Buchberger. An algorithmic criterion for the solvability of algebraic systems of equations. In B. Buchberger and F. Winkler, editors, *Gröbner Bases and Applications, 33 Years of Gröbner Bases*, London Mathematical Society Lecture Notes Series 251. Cambridge University Press, 1998.
3. Bruno Buchberger. Lifting knowledge to the state of inferencing. Technical Report TR 2004-12-03, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, 2004.
4. Bruno Buchberger. Proving by first and intermediate principles, November 2, 2004. Invited talk at Workshop on Types for Mathematics / Libraries of Formal Mathematics, University of Nijmegen, The Netherlands.
5. Bruno Buchberger, Adrian Crăciun, Tudor Jebelean, Laura Kovács, Temur Kutsia, Koji Nakagawa, Florina Piroi, Nikolaj Popov, Judit Robu, Markus Rosenkranz, and Wolfgang Windsteiger. Theorema: Towards computer-aided mathematical theory exploration. *Journal of Applied Logic*, pages 470–504, 2006.
6. Martin Giese and Bruno Buchberger. Towards practical reflection for formal mathematics. Technical Report 07-05, RISC, Johannes Kepler University, 2007.
7. Warren A. Hunt Jr., Matt Kaufmann, Robert Bellarmine Krug, J Strother Moore, and Eric Whitman Smith. Meta reasoning in ACL2. In Joe Hurd and Thomas F. Melham, editors, *Proc. Theorem Proving in Higher Order Logics, TPHOLs 2005, Oxford, UK*, volume 3603 of *LNCS*, pages 163–178. Springer, 2005.
8. Temur Kutsia and Bruno Buchberger. Predicate logic with sequence variables and sequence function symbols. In A. Asperti, G. Bancerek, and A. Trybulec, editors, *Proc. 3rd Intl. Conf. on Mathematical Knowledge Management, MKM'04*, volume 3119 of *LNCS*, pages 205–219. Springer Verlag, 2004.
9. Stephen Wolfram. *The Mathematica Book*. Wolfram Media, Inc., 1996.