Towards Practical Reflection for Formal Mathematics

Martin Giese[1] and Bruno Buchberger[2]

[1] RICAM, Austrian Academy of Sciences,
Altenbergerstr. 69, A-4040 Linz, Austria
martin.giese@oeaw.ac.at
[2] RISC, Johannes Kepler University,
A-4232 Schloß Hagenberg, Austria
bruno.buchberger@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.

1 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 [5, 6], and as part of an ongoing effort to redesign and reimplement the Theorema system [7], 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 describe our approach as it has been implemented by the first author in a prototype called Mini-Tma, a Mathematica [18] 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.

The second author's contributions are the identification and formulation of the problem addressed in this paper and the recognition of its importance for mathematical theory exploration [6], as well as a first illustrating example [5], a simplified version of which will be used in this paper. The first author has worked out the technical details and produced the implementation of Mini-Tma.

In Sect. 2, we introduce the required concepts on the level of the system's logical language. Sect. 3 shows how this language can be used to describe new reasoners, and how they can be applied. Sect. 4 illustrates how the system can be used to reason about the logic itself. These techniques are combined in Sect. 5 to reason about reasoners. We briefly disuss some foundational issues in Sect. 6. Related work is reviewed in Sect. 7, and Sect. 8 concludes the paper.

2 The Framework

To reason about the syntactic (terms, formulae, proofs,...) and semantic (models, validity...) concepts that constitute a logic, it is in principle sufficient to axiomatize these concepts, which is possible in any logic that permits e.g. inductive data type definitions, and reasoning about them. This holds also if the formalized logic is the same as the logic it is being formalized in, which is the case that interests us here.

However, to make this reasoning *attractive* enough to become a natural part of using a mathematical assistant system, we consider it important to supply a *built-in* representation of at least the relevant syntactic entities. In other words, one particular way of expressing statements about terms, formulae, etc. needs to be chosen, along with an appealing syntax, and made part of the logical language.

We start from the logic previously employed in the Theorema system, namely an untyped higher-order predicate logic with sequence variables. Sequence variables [16] represent sequences of values and have proven to be very convenient for expressing statements about operations with variable arity. For instance, the operation app that appends two lists can be specified by[1]

$$\underset{\overline{xs}}{\forall}\,\underset{\overline{ys}}{\forall}\,\mathsf{app}[\{\overline{xs}\},\{\overline{ys}\}]=\{\overline{xs},\overline{ys}\}$$

using two sequence variables $\overline{xs}$ and $\overline{ys}$. It turns out that sequence variables are also convenient in statements about terms and formulae, since term construction in our logic is a variable arity operation.

2.1 Quoting

Terms in our logic are constructed in two ways: *symbols* (constants or variables) are one kind of terms, and the other are *compound terms*, constructed by 'applying' a 'head' term to a number of 'arguments'.[2] 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. Quoted symbols are themselves symbols, so there are quoted versions of them too, i.e. if a is in the signature, then so are $\underline{\mathsf{a}}$, $\underline{\underline{\mathsf{a}}}$, etc. For compound terms, the obvious representation would have been a dedicated term construction function, say mkTerm, such that $\mathsf{f}[\mathsf{a}]$ would be denoted by $\mathsf{mkTerm}[\underline{\mathsf{f}},\underline{\mathsf{a}}]$. Using a special syntax, e.g. fancy brackets, would have allowed us to write something like $\underline{\mathsf{f}}\{\underline{\mathsf{a}}\}$. However, experiments revealed that (in an untyped logic!) it is easiest to reuse the function application brackets $[\cdots]$ for term construction and requiring that if whatever stands to the left of the brackets is a term, then term construction instead of function application is meant. Any axioms or reasoning rules involving term construction contain this condition on the head term. This allows us to write $\underline{\mathsf{f}}[\underline{\mathsf{a}}]$, which is easier to read and easier to input to the system. For reasoning, the extra condition that the head needs to be a term is no hindrance, since this condition usually has to be dealt with anyway.

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 instance, $\underline{\mathsf{f}[\mathsf{a},\mathsf{h}[\mathsf{b}]]}$ is accepted as shorthand for $\underline{\mathsf{f}}[\underline{\mathsf{a}},\underline{\mathsf{h}}[\underline{\mathsf{b}}]]$. The system will also output quoted terms in this fashion whenever possible. While this is convenient, it is important to understand that it is just a nicer presentation of the underlying representation that requires only quoting of symbols and complex term construction as function application.

[1] Following the notation of Mathematica and Theorema, we use square brackets $[\cdots]$ to denote function application throughout this paper. Constant symbols will be set in $\mathsf{sans\text{-}serif}$ type, and variable names in *italic*.

[2] See Sect. 2.2 for the issue of variable binding in quantifiers, lambda terms, and such.

2.2 Dealing with Variable Binding

In the literature, various thoughts can be found on how to appropriately represent variable binding operators, i.e. quantifiers, lambda abstraction, etc. The dominant approaches are 1. higher-order abstract syntax, 2. de Bruijn indices, and 3. explicit representation.

Higher-order abstract syntax (HOAS) [17] is often used to represent variable binding in logical frameworks and other systems built on higher-order logic or type theory. With HOAS, a formula $\underset{x}{\forall}\, \mathsf{p}[x]$ would be represented as $\underline{\mathsf{ForAll}}[\underset{\xi}{\lambda}\, \underline{\mathsf{p}}[\xi]]$. The argument of the $\underline{\mathsf{ForAll}}$ symbol is a function which, for any term ξ delivers the result of substituting ξ for the bound variable x in the scope of the quantifier. This representation has its advantages, in particular that terms are automatically stored modulo renaming of bound variables, and that capture-avoiding substitution comes for free, but we found it to be unsuitable for our purposes: some syntactic operations, such as comparing two terms for syntactic equality are not effectively possible with HOAS, and also term induction, which is central for reasoning about logics, is not easily described. Hendriks has come to the same conclusion in his work on reflection for Coq [13].

Hendriks uses de Bruijn indices [10], which would represent $\underset{x}{\forall}\, \mathsf{p}[x]$ by a term like $\underline{\mathsf{ForAll}}[\underline{\mathsf{p}}[v_1]]$, where v_1 means the variable bound by the innermost binding operator, v_2 would mean to look one level further out, etc. This representation has some advantages for the implementation of term manipulation operations and also for reflective reasoning about the logic.

For Mini-Tma however, in view of the projected integration of our work into the Theorema system, we chose a simple explicit representation. The reason is mainly that we wanted the representations to be as readable and natural as possible, to make it easy to debug reasoners, to use them in interactive theorem proving, etc. A representation that drops the names of variables would have been disadvantageous. 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 reason for having only one binding operator is to be able to describe operations like capture avoiding substitution without explicitly naming all operators that might bind a variable. Under this convention, we consider the effort of explicitly dealing with α-conversion to be acceptable: the additional difficulty appears mostly in a few basic operations on terms, which can be implemented once and for all, after which there is no longer any big difference between the various representations.

2.3 An Execution Mechanism

Writing and verifying programs has always been part of the Theorema project's view of mathematical theory exploration [15]. It is also important in the context of this paper, since we want users of the system to be able to define new reasoners, meaning programs that act on terms.

In order to keep the system's input language as simple and homogenous as possible, we use its logical language as programming language. Instead of fixing any particular way of interpreting formulae as programs, Mini-Tma supports the general concept of *computation mechanisms.* Computations are invoked from the user interface by typing[3]

$$\text{Compute}[\mathit{term}, \text{by} \to \mathit{comp}, \text{using} \to \mathit{ax}]$$

where *term* is the term which should be evaluated, *comp* names a computation mechanism, and *ax* is a set of previously declared axioms. Technically, *comp* is a function that is given *term* and *ax* as arguments, and which eventually returns a term. The intention is that *comp* should compute the value of *term*, possibly controlled by the formulae in *ax*. General purpose computation mechanisms require the formulae of *ax* to belong to a well-defined subset of predicate logic, which is interpreted as a programming language. A special purpose computation mechanism might e.g. only perform arithmetic simplifications on expressions involving concrete integers, and completely ignore the axioms. In principle, the author of a computation mechanism has complete freedom to choose what to do with the term and the axioms.

We shall see in Sect. 3 that it is possible to define new computation mechanisms in Mini-Tma. It is however inevitable to provide at least one built-in computation mechanism which can be used to define others. This 'standard' computation mechanism of Mini-Tma is currently based on conditional rewriting. It requires the axioms to be equational Horn clauses.[4] Program execution proceeds by interpreting these Horn clauses as conditional rewrite rules, applying equalities from left to right. Rules are exhaustively applied innermost-first, and left-to-right, and applicability is tested in the order in which the axioms are given. The conditions are evaluated using the same computation mechanism, and all conditions have to evaluate to True for a rule to be applicable. The system does not order equations, nor does it perform completion. Termination and confluence are in the responsibility of the programmer.

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.

3 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

[3] Compute, by, using are part of the *User Language*, used to issue commands to the system. Keywords of the User Language will by set in a serif font.

[4] Actually, for convenience, a slightly more general format is accepted, but it is transformed to equational Horn clauses before execution.

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.

Consider for instance an exploration of the theory of natural numbers. After the associativity of addition has been proved, and used to prove several other theorems, we notice that it is always possible to rewrite terms in such a way that all sums are grouped to the right. Moreover, this transformation is often useful in proofs, since it obviates most explicit applications of the associativity lemma. This suggests implementing a new computation mechanism that transforms terms containing the operator Plus in such a way that all applications of Plus are grouped to the right. E.g., we want to transform the term $\mathsf{Plus}[\mathsf{Plus}[\mathsf{a},\mathsf{b}],\mathsf{Plus}[\mathsf{c},\mathsf{d}]]$ to $\mathsf{Plus}[\mathsf{a},\mathsf{Plus}[\mathsf{b},\mathsf{Plus}[\mathsf{c},\mathsf{d}]]]$, ignoring any axioms. We start by defining a function that will transform *representations* of terms, e.g. $\underline{\mathsf{Plus}}[\underline{\mathsf{Plus}}[\underline{\mathsf{a}},\underline{\mathsf{b}}],\underline{\mathsf{Plus}}[\underline{\mathsf{c}},\underline{\mathsf{d}}]]$ to $\underline{\mathsf{Plus}}[\underline{\mathsf{a}},\underline{\mathsf{Plus}}[\underline{\mathsf{b}},\underline{\mathsf{Plus}}[\underline{\mathsf{c}},\underline{\mathsf{d}}]]]$. We do this with the following definition:

$$
\begin{array}{l}
\text{Axioms}\,[\text{"shift parens"},\text{any}[s,t,t_1,t_2,acc,l,ax,comp],\\
\quad \mathsf{simp}[t,ax,comp]=\mathsf{add\text{-}terms}[\mathsf{collect}[t,\{\}]]\\[4pt]
\quad \mathsf{collect}\,[\underline{\mathsf{Plus}}[t_1,t_2],acc]=\mathsf{collect}[t_1,\mathsf{collect}[t_2,acc]]\\
\quad \mathsf{is\text{-}symbol}[t]\Rightarrow\mathsf{collect}[t,acc]=\mathsf{cons}[t,acc]\\
\quad \mathsf{head}[t]\neq\underline{\mathsf{Plus}}\Rightarrow\mathsf{collect}[t,acc]=\mathsf{cons}[t,acc]\\[4pt]
\quad \mathsf{add\text{-}terms}[\{\}]=\underline{\mathsf{0}}\\
\quad \mathsf{add\text{-}terms}[\mathsf{cons}[t,\{\}]]=t\\
\quad \mathsf{add\text{-}terms}[\mathsf{cons}[s,\mathsf{cons}[t,l]]]=\underline{\mathsf{Plus}}[s,\mathsf{add\text{-}terms}[\mathsf{cons}[t,l]]]\\
]
\end{array}
$$

The main function is simp, its arguments are the term t, the set of axioms ax, and another computation mechanism $comp$, which will be explained later. simp performs its task by calling an auxiliary function $\mathsf{collect}$ which recursively collects the fringe of non-Plus subterms in a term, prepending them to an accumulator acc that is passed in as second argument, and that starts out empty. To continue our example, $\mathsf{collect}[\underline{\mathsf{Plus}}[\underline{\mathsf{Plus}}[\underline{\mathsf{a}},\underline{\mathsf{b}}],\underline{\mathsf{Plus}}[\underline{\mathsf{c}},\underline{\mathsf{d}}]],\{\}]$ evaluates to the list of (quoted) terms $\{\underline{\mathsf{a}},\underline{\mathsf{b}},\underline{\mathsf{c}},\underline{\mathsf{d}}\}$. This list is then passed to a second auxiliary function $\mathsf{add\text{-}terms}$ which builds a $\underline{\mathsf{Plus}}$-term from the elements of a list, grouping to the right. Note that this transformation is done completely without reference to rewriting or the associativity lemma. We are interested in programs that can perform arbitrary operations on terms.

The function $\mathsf{is\text{-}symbol}$ is evaluated to True if its argument represents a symbol and not a complex term or any other object. This and some other operations (equality of terms, ...) are handled by built-in rewriting rules since a normal axiomatization would not be possible, or in some cases too inefficient.

Given these axioms, we can now ask the system to simplify a term:

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

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

So far, this is an example of a computation that works on terms, and not very different from a computation on, say, numbers. But we can now make simp known to the system as a computation mechanism. After typing

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

the system recognizes a new computation mechanism named ShiftParens. We can now tell it to

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

and receive the answer $\mathsf{Plus}[\mathsf{a}, \mathsf{Plus}[\mathsf{b}, \mathsf{Plus}[\mathsf{c}, \mathsf{d}]]]$. No more quotation is needed, the behavior is just like for any built-in computation mechanism. Also note that no axioms need to be given, since the ShiftParens computation mechanism does its job without considering the axioms.

We now come back to the extra argument *comp*: Mini-Tma allows computation mechanisms to be combined in various ways, which we shall not discuss in this paper, in order to obtain more complex behavior. However, even when actual computations are done by different mechanisms, within any invocation of Compute, there is always one *global computation mechanism*, which is the top-level one the user asked for. It happens quite frequently that user-defined computation mechanisms would like to delegate the evaluation of subterms that they cannot handle themselves to the global computation mechanism. It is therefore provided as the argument *comp* to every function that is used as a computation mechanism, and it can be called like a function.

Calling a user-defined computation mechanism declared to be implemented as a function simp on a term t with some axioms ax under a global computation mechanism $comp$ proceeds as follows: 1. t is quoted, i.e. a term t' is constructed that represents t, 2. $\mathsf{simp}[t', ax, comp]$ is evaluated using the computation mechanism and axioms fixed in the DeclareComputer invocation. 3. The result s' should be the representation of a term s, and that s is the result. If step 2 does not yield a quoted term, an error is signaled.

The ShiftParens simplifier is of course a very simple example, but the same principle can clearly be used to define and execute arbitrary syntactic manipulations, including proof search mechanisms within the system's logical language. Since most reasoning algorithms proceed by applying reasoning rules to some proof state, constructing a proof tree, the Theorema implementation will include facilities that make it easy to express this style of algorithm, which would be more cumbersome to implement in out prototypical Mini-Tma system.

4 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—there is no type system that enforces the arities of function applications—and arbitrary terms can appear as the heads of complex terms. Sequence variables are very convenient in dealing with the variable length argument lists. While axiomatizing operations like capture avoiding substitution on arbitrary term representations, we employed a recursion scheme based on the observation that a term is either a symbol, or a complex term with an empty argument list, or the result of adding an extra argument to the front of the argument list of another complex term, or a lambda abstraction. The corresponding induction rule is:[5]

$$\frac{\begin{array}{l} \underset{\mathsf{is\text{-}symbol}[s]}{\forall} P[s] \\ \underset{\mathsf{is\text{-}term}[f]}{\forall} (P[f] \Rightarrow P[f[]]) \\ \underset{\substack{\mathsf{is\text{-}term}[f]\\ \mathsf{is\text{-}term}[hd]\\ \mathsf{are\text{-}terms}[\overline{tl}]}}{\forall} (P[hd] \wedge P[f[\overline{tl}]] \Rightarrow P[f[hd,\overline{tl}]]) \\ \underset{\substack{\mathsf{is\text{-}term}[t]\\ \mathsf{is\text{-}symbol}[x]}}{\forall} (P[t] \Rightarrow P[\underline{\lambda}[x,t]]) \end{array}}{\underset{\mathsf{is\text{-}term}[t]}{\forall} P[t]}$$

Using the mechanism outlined in Sect. 3, we were able to implement a simple term induction prover, that applies the term induction rule once, and then tries to prove the individual cases using standard techniques (conditional rewriting and case distinction), in less than 1000 characters of code. This naïve prover is sufficient to prove simple statements about terms, like e.g.

$$\underset{\substack{\mathsf{is\text{-}term}[t]\\ \mathsf{is\text{-}symbol}[v]\\ \mathsf{is\text{-}term}[s]}}{\forall} (\mathsf{not\text{-}free}[t,v] \Rightarrow t\{v \to s\} = t)$$

where $\mathsf{not\text{-}free}[t,v]$ denotes that the variable v does not occur free in t, and $t\{v \to s\}$ denotes capture avoiding substitution of v by s in t, and both these notions are defined through suitable axiomatizations.

5 Reasoning About Reasoners

Program verification plays an important role in the Theorema project [15]. Using predicate logic as a programming language obviously makes it particularly easy to reason about programs' partial correctness. Of course, termination has to be proved separately.

With Mini-Tma's facilities for writing syntax manipulating programs, and for reasoning about syntactic entities, it should come as no surprise that it is

[5] $\underset{p[x]}{\forall} q[x]$ is just convenient syntax for $\underset{x}{\forall} (p[x] \Rightarrow q[x])$

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.

So far, we have concentrated mainly on soundness proofs. In the literature, we have found two ways of proving the soundness of reasoners: the first way consists in proving that the new reasoner cannot *prove* anything that cannot be proved by the existing calculus. Or, in the case of a simplifier like ShiftParens of Sect. 3, if a simplifier simplifies t to t', then there is a *rewriting* proof between t and t'. This approach is very difficult to follow in practice: it requires formalizing the existing calculus, including proof trees, possibly rewriting, etc. Often the soundness of a reasoner will depend on certain properties of the involved operations, e.g. ShiftParens requires the associativity of Plus, so the knowledge base has to be axiomatized as well. Moreover, to achieve reasonable proof automation, the axiomatization needs to be suitable for the employed prover: finding a proof can already be hard, making prover A *prove* that prover B will find a proof essentially requires re-programming B in the axiomatization. And finally, this correctness argument works purely on the syntactic level: any special reasoning techniques available for the mathematical objects some reasoner is concerned with are useless for its verification!

We have therefore preferred to investigate a second approach: we prove that anything a new reasoner can prove is simply *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 very successful NqThm and ACL2 systems [2, 14]. It solves the above problems, since it is a lot easier to axiomatize a model semantics for our logic, and the axiomatization is also very easy to use for an automated theorem prover. The knowledge base does not need to be 'quoted', since much of the reasoning is about the values instead of the terms, and for the same reason, any previously implemented special reasoners can be employed in the verification.

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.[6] 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 $\beta[\underline{\mathsf{0}}] = \mathsf{0}$ and $\beta[\underline{\mathsf{Plus}}] = \mathsf{Plus}$. To prove this statement inductively, it needs to be strengthened to

$$\mathsf{eval}[\mathsf{add\text{-}terms}[\mathsf{collect}[t, acc]], \beta] = \mathsf{eval}[\underline{\mathsf{Plus}}[t, \mathsf{add\text{-}terms}[acc]], \beta] \qquad (*)$$

for any acc, and an additional lemma

$$\mathsf{eval}[\mathsf{add\text{-}terms}[\mathsf{cons}[t, l]], \beta] = \mathsf{Plus}[\mathsf{eval}[t, \beta], \mathsf{eval}[\mathsf{add\text{-}terms}[l], \beta]]$$

[6] Care needs to be taken when applying eval to terms containing $\underline{\mathsf{eval}}$, as has already been recognized by Boyer and Moore [3].

is required. And of course, the associativity of Plus needs to known. Mini-Tma cannot prove (∗) with the term induction prover described in Sect. 4, since it is not capable of detecting the special role of the symbol Plus. However, using a modified induction prover which treats compound terms with head symbol Plus as a separate case, (*) can be proved automatically.

Automatically extracting such case distinctions from a program is quite conceivable, and one possible topic for future work on Mini-Tma.

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 [4],
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 hope to have convinced the reader that the outlined approach can be extended to more complex applications.

6 Foundational Issues

Most previous work on reflection in theorem proving environments (see Sect. 7) has concentrated on the subtle foundational problems arising from adding reflection to an existing system. In particular, any axiomatization of the fact that a reflectively axiomatized logic behaves exactly like the one it is being defined in can easily lead to inconsistency. In our case, care needs to be taken with the evaluation function eval which connects the quoted logic to the logic it is embedded in.

However, within the Theorema project, we are not particularly interested in the choice and justification of a single logical basis. Any framework a mathematician considers appropriate for the formalization of mathematical content should be applicable within the system—be it one or the other flavor of set theory, type theory, or simply first-order logic. Any restriction to one particular framework would mean a restriction to one particular view of mathematics, which is something we want to avoid. This is why there is no such thing as *the* logic of Theorema. But if there is no unique, well-defined basic logic, then neither can we give a precise formal basis for its reflective extension. In fact, since the way in which such an extension is defined is itself an interesting mathematical subject, we do not even want to restrict ourselves to a single way of doing it.

This is of course somewhat unsatisfying, and it is actually not the whole truth. We *are* trying to discover a particularly viable *standard* method of adding reflection and reflective reasoners. And we are indeed worried about the soundness of that method. It turns out that one can convince oneself of the soundness of such an extension provided the underlying logic satisfies a number of reasonable assumptions.

Let a logical language $\mathcal{L}$ be given. In the context of formalization of mathematics, we may assume that syntactically, $\mathcal{L}$ consists of a subset of the formulae of higher order predicate logic. Typically, some type system will forbid the construction of certain ill-typed formulae, maybe there is also a restriction to first-order formulae.

Most logics permit using a countably infinite signature, in fact, many calculi require the presence of infinitely many constant symbols for skolemization. Adding a quoted symbol $\underline{\mathsf{a}}$ for any symbol a of $\mathcal{L}$ will then be unproblematic.

Next, we can add a function $\mathsf{is\text{-}symbol}$, which may be defined through a countably infinite and effectively enumerable family of axioms, which should not pose any problems. The function $\mathsf{is\text{-}term}$ can then be axiomatized recursively in any logic that permits recursive definitions. We can assume for the moment that the logic does not include quoting for $\mathsf{is\text{-}symbol}$ or $\mathsf{is\text{-}term}$, and that the functions will recognize the quotations of symbols and terms of $\mathcal{L}$, and not of the reflectiove extension of $\mathcal{L}$ we are constructing.

Likewise, if the evaluation of basic symbols is delegated to an assignment β, it should be possible to give an axiomatization of the recursive evaluation function eval within any logic that permits recursive definitions:

$$\mathsf{is\text{-}symbol}[t] \Rightarrow \mathsf{eval}[t, \beta] = \beta[t]$$
$$\mathsf{is\text{-}term}[f] \Rightarrow \mathsf{eval}[f[t], \beta] = \mathsf{eval}[f, \beta][\mathsf{eval}[t, \beta]]$$

The exact definitions will depend on the details of $\mathcal{L}$. For instance, if $\mathcal{L}$ is typed, it might be necessary to introduce a family of eval functions for terms of different types, etc. Still, we do not believe that soundness problems can occur here.

The interesting step is now the introduction of an unquoting function unq, which relates every quoted symbol $\underline{\mathsf{a}}$ to the entity it represents, namely a. We define unq by the axioms

$$\mathsf{unq}[\mathbf{s}'] = \mathbf{s}$$

for all symbols $\mathbf{s}$ of $\mathcal{L}$, where $\mathbf{s}'$ denotes the result of applying one level of reflection quoting to $\mathbf{s}$, i.e. $\mathsf{unq}[\underline{\mathsf{a}}] = \mathsf{a}$, $\mathsf{unq}[\underline{\mathsf{b}}] = \mathsf{b}$,... The formula $\mathsf{unq}[\underline{\mathsf{unq}}] = \mathsf{unq}$ is *not* an axiom since this would precisely lead to the kind of problems identified by Boyer and Moore in [3]. If they are only present for the symbols of the original logic, these axioms do not pose any problems.

All in all, the combined extension is then a conservative extension of the original logic, meaning that any model $\mathcal{M}$ for a set Φ of formulae of $\mathcal{L}$ can be extended to a model $\mathcal{M}'$ of Φ in the reflective extension, such that $\mathcal{M}'$ behaves like $\mathcal{M}$ when restricted to the syntax of $\mathcal{L}$. Moreover, in the extension, the formula

$$\mathsf{eval}[\mathbf{t}', \mathsf{unq}] = \mathbf{t}$$

holds for every term $\mathbf{t}$ of $\mathcal{L}$ with quotation $\mathbf{t}'$, which justifies using eval to prove the correctness of new reasoners.

To allow for several levels of quotation, this process can be iterated. It is easy to see that the $\mathsf{is\text{-}symbol}$, $\mathsf{is\text{-}term}$, and eval functions defined for consecutive levels can be merged. For the unq function, one possible solution is to use a hierarchy $\mathsf{unq}^{(i)}$ of unquoting functions, where there is an axiom $\mathsf{unq}^{(i)}[\underline{\mathsf{unq}^{(j)}}] = \mathsf{unq}^{(j)}$ if and only if $j < i$.

Another difficulty is the introduction of *new* symbols required by many calculi for skolemization, which can be jeopardized by the presence of knowledge about the unquoting of quoted symbols. Here, a possible solution is to fix the set of symbols for which unq axioms are required before proofs, as is done in ACL2.

7 Related Work

John Harrison has written a very thorough survey [11] of reflection mechanisms in theorem proving systems, and most of the work reviewed there is in some way connected to ours.

The most closely related approach is surely that of the NqThm and ACL2 systems, see e.g. [2, 14]. The proving power of these systems can be extended by writing simplifiers in the same programming language as that which can be verified by the system. Before using a new simplifier, its soundness has to be shown using a technique similar to that of Sect. 5. Our work extends theirs in the following respects:

– We use a stronger logic, ACL2 is restricted to first-order quantifier-free logic.
– Our framework allows coding full, possibly non-terminating theorem provers, and not just simplifiers embedded in a fixed prover.
– Through the *comp* argument, reasoners can be called recursively.
– The specialized quoting syntax and sequence variables make Mini-Tma more pleasant and practical to use.
– In Mini-Tma, Meta-programming can be used without being forced to prove soundness first, which is useful for experimentation and exploration.

Experiments in reflection have also recently been done in Coq [13], but to our knowledge these are restricted to first-order logic, and meta-programmed provers cannot be used as part of a proof construction. There has also been some work on adding reflection to Nuprl [1]. This is still in its beginnings, and its principal focus seems to be to prove theorems about logics, while our main goal is to increase the system's reasoning power.

Recent work on the self-verification of HOL Light [12] is of a different character. Here, the HOL Light system is not used to verify extensions of itself, but rather for the self-verification of the kernel of the system. Self-verification raises some foundational issues of its own that do not occur in our work.

In the context of programming languages, LISP has always supported quoting of programs and meta-programming, e.g. in macros. Amongst more modern languages, Maude should be mentioned for its practically employed reflective capabilities, see e.g. [9]. A quoting mechanism is part of the language, and it is used to define the 'full' Maude language in terms of a smaller basic language. However, this reflection is just used for programming, there is no reasoning involved.

8 Conclusion and Future Work

We have reported on ongoing research in the frame of the Theorema project, that aims at making coding of new (special purpose) reasoners and reasoning about them, e.g. to prove their soundness, an integral part of the theory exploration process within the system. The approach has been evaluated in the prototypical implementation Mini-Tma.

The main features of our approach are to start from a logic with a built-in quoting mechanism, and to use the same logic for the definition of programs, and in particular reasoners. We have shown that this makes it possible to define reasoners which can be used by the system like the built-in ones. It also enables the user to reason about terms, formulae, etc. and also about reasoners themselves.

We have briefly discussed two alternatives for defining and proving the soundness of new reasoners, and concluded that an approach based on formalizing a model semantics is more suitable for automated deduction than one that is based on formalizing proof theory.

Future work includes improving the execution efficiency of programs written within the Theorema logic. Improvements are also required for the theorem proving methods, i.e. better heuristics for term induction, program verification, etc., but also the production of human-readable proof texts or proof trees, which are essential for the successful application of the theorem provers. All these developments will have to be accompanied by case studies demonstrating their effectiveness.

Acknowledgments

The authors would like to thank the other members of the Theorema Group, in particular Temur Kutsia and Markus Rosenkranz, for contributing to the numerous intense discussions about the presented work.

References

1. Eli Barzilay. *Implementing Reflection in Nuprl.* PhD thesis, Cornell University Computer Science, 2006.
2. Robert S. Boyer, Matt Kaufmann, and J Strother Moore. The Boyer-Moore theorem prover and its interactive enhancement. *Computers and Mathematics with Applications*, 29(2):27–62, 1995.
3. Robert S. Boyer and J Strother Moore. The addition of bounded quantification and partial functions to a computational logic and its theorem prover. *J. Autom. Reasoning*, 4(2):117–172, 1988.

4. Bruno Buchberger. Ein algorithmisches Kriterium für die Lösbarkeit eines algebraischen Gleichungssystems. *Aequationes Math.*, 4:374–383, 1970. English translation published in [8].
5. 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.
6. 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.
7. 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.
8. Bruno Buchberger and Franz Winkler. Gröbner bases and applications. In B. Buchberger and F. Winkler, editors, *33 Years of Gröbner Bases*, London Mathematical Society Lecture Notes Series 251. Cambridge University Press, 1998.
9. Manuel Clavel and José Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285(2):245–288, 2002.
10. Nicolas G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 34:381–392, 1972.
11. John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995.
12. John Harrison. Towards self-verification of HOL Light. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *LNCS*, pages 177–191. Springer, 2006.
13. Dimitri Hendriks. Proof reflection in Coq. *Journal of Automated Reasoning*, 29(3–4):277–307, 2002.
14. 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.
15. Laura Kovács, Nikolaj Popov, and Tudor Jebelean. Verification environment in Theorema. *Annals of Mathematics, Computing and Teleinformatics (AMCT)*, 1(2):27–34, 2005.
16. 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.
17. Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proc. ACM SIGPLAN 1988 Conf. on Programming Language design and Implementation, PLDI '88, Atlanta, United States*, pages 199–208. ACM Press, New York, 1988.
18. Stephen Wolfram. *The Mathematica Book*. Wolfram Media, Inc., 1996.