Interactive Proving, Higher-Order Rewriting, and Theory Analysis in Theorema 2.0

Alexander Maletzky*

Doctoral Program “Computational Mathematics” and RISC
Johannes Kepler University Linz, Austria
alexander.maletzky@dk-compmath.jku.at,
https://www.dk-compmath.jku.at/people/alexander-maletzky/

Abstract. In this talk we will report on three useful tools recently implemented in the frame of the Theorema project: a graphical user interface for interactive proof development, a higher-order rewriting mechanism, and a tool for automatically analyzing the logical structure of Theorema-theories. Each of these three tools already proved extremely useful in the extensive formal exploration of a non-trivial mathematical theory, namely the theory of Gröbner bases and reduction rings, in Theorema 2.0.

Keywords: Computer-assisted mathematical theory exploration, interactive theorem proving, Theorema

1 Introduction

Theorema[1] is a so-called *mathematical assistant system* supporting its users in all aspects of mathematical theory exploration: inventing new notions and problems, implementing and experimenting with algorithms, making conjectures, and finally proving or disproving them. Theorema 2.0 [1] is the latest version of the system, released roughly two years ago in 2014; as its predecessor, it is still based on *Mathematica*.

The present paper reports on three tools we recently developed for making working with the system more attractive and efficient: a versatile *interactive proof strategy* giving the user full control over proving and complementing the existing automatic strategies, a powerful *rewriting mechanism* for translating first- and higher-order formulas into *Mathematica* transformation rules for rewriting other formulas in proofs, and a simple but nonetheless extremely helpful tool for analyzing the logical structure of Theorema-theories. The development of each of these three tools was motivated by our extensive formal treatment of the theory of Gröbner bases and reduction rings in Theorema, see [3] for details.

Please note that the tools have not been integrated into the official version of Theorema 2.0 yet, but they are expected to be in the near future. Still, they

* This research was funded by the Austrian Science Fund (FWF): grant no. W1214-N15, project DK1

[1] http://www.risc.jku.at/research/theorema/software/

can easily be installed manually, relying on *Mathematica*'s comfortable package-system.

Small parts of this paper are also contained in [3].

2 Interactive Proving

The first of the three tools we present in this paper is an *interactive proof strategy* (IPS) that, as its name suggests, can be used for developing proofs in Theorema 2.0 fully interactively, in the sense that the human user has full control over what happens at each stage of a proof. This is in contrast to the automatic, or, at least, semi-automatic proof strategies typically available in the system.

As can be seen in a concrete example below, the IPS in Theorema 2.0 is not text-based, as in most other proof assistants, but *dialog-oriented.* This means that whenever a user interaction is required, a dialog window displaying the current proof situation pops up, asking the user to perform an action (by clicking on a button, typing in some text, etc.; see Fig. 1). This, in fact, follows the tradition of interactive proving in Theorema 1, the predecessor version of Theorema 2.0, where the environment for interactive proving developed in [5] is dialog-oriented as well. Note that we did not just migrate said environment from Theorema 1 to Theorema 2.0, but really implemented the new IPS completely from scratch; this seemed to be the more reasonable approach, as the internal architecture of Theorema 2.0 differs considerably from the one of Theorema 1.

Before explaining how the IPS can be used in practical applications, some words on its implementation are in place: the IPS is implemented simply as a Theorema proof strategy, meaning that it essentially is a function taking a *proof situation* (characterized by the current proof goal and a list of assumptions) as input and returning a list of new, ideally simpler proof situations as output; the logical relation between in- and output obviously is that the validity of the input-situation follows from the validity of all of the output-situations. The output is constructed by applying *inference rules* that are themselves independent of the IPS and could well be used together with any other (automatic) proof strategy installed in the system. The main task of the IPS is only to *guide* the application of the inference rules, by specifying which rules shall be applied and how they shall be applied.

2.1 How the Interactive Proof Strategy Works in Practice

Once the IPS is properly installed, it can be selected as the proof strategy of choice just as any other, pre-defined proof strategy when initiating a proof attempt; no further setup by the user is required. Then, whenever a new proof situation p arises during the proof search, the IPS proceeds as follows:

- First, it automatically tries to apply an available *high-priority* inference rule to p. If this is possible, the respective rule is applied and the proof search continues.

– Otherwise, if no high-priority rule is applicable to p, it asks the user how to proceed by displaying a graphical dialog window.

Every inference rule in Theorema has a *priority* attached to it. Automatic proof strategies usually fall back on these priorities for determining the order they try to apply inference rules in. The IPS takes rule priorities into account solely for filtering out the high-priority rules, i. e. those rules whose priorities are above a certain, user-adjustable threshold.[2]

Assume now that no high-priority rule could be applied to p. The user now has a range of possibilities how to proceed, including

– choosing another inference rule to apply to p (or, more precisely, to *try*, since non-applicable rules are not automatically filtered out),
– choosing a different pending proof situation where to continue,
– adjusting various settings, like the current set of inference rules and even the proof strategy (making it possible to switch to an automatic strategy at some point during the proof development),
– inspecting the so-far constructed proof in a nicely-formatted proof document,
– inspecting the internal representation of p as a plain *Mathematica* expression for debugging purposes,
– saving the current proof status to an external file, for creating a "secure point" the proof may be resumed from later, and
– aborting the proof attempt.

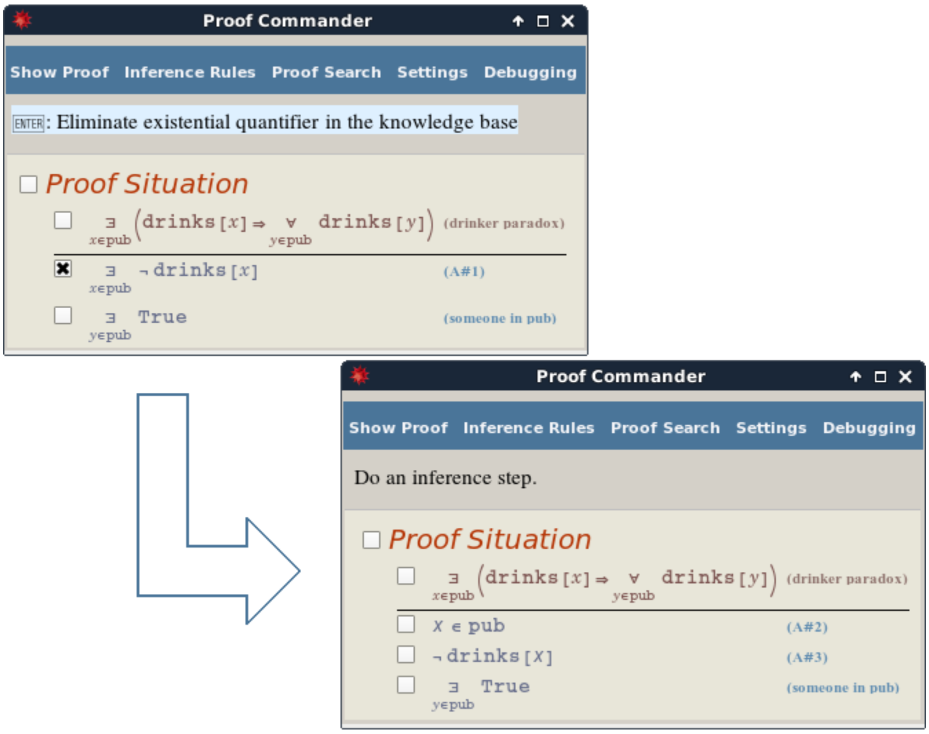
Before choosing an inference rule the user may also activate and deactivate formulas appearing in p by marking check-boxes in the dialog window (see Fig. 1). This might affect *how* the chosen rule is applied, e. g. if several cases based on a disjunction in the knowledge base shall be distinguished, but more than one disjunctions appear among the assumptions, the user can specify exactly which one to consider simply by deactivating all others. It must be noted, though, that the information about whether a formula is activated or not might well be ignored by the chosen inference rule; this cannot be influenced by the IPS.

2.2 An Example

As an example, let us consider the interactive proof of the well-known *drinker paradox*: "In every non-empty pub there is someone such that, if he is drinking, everyone else is drinking as well." This is actually no paradox but a theorem in classical logic and may hence be proved in Theorema.

Figure 1 depicts two dialog windows of the IPS arising in the interactive proof of the drinker paradox. The first one corresponds to the case where someone who does *not* drink is assumed to be in the pub (Formula (A#1)), and where the next action to be taken, as specified by the user, is to eliminate the existential quantifier in Formula (A#1) by introducing a new constant that witnesses this person. The resulting proof situation is displayed in the second window.

[2] A typical example of a high-priority rule is the inference rule that proves implications by assuming their premises and proving their conclusions.



ENTER: Eliminate existential quantifier in the knowledge base

□ Proof Situation

□ $\exists_{x \in \mathtt{pub}} \left(\mathtt{drinks}[x] \Rightarrow \forall_{y \in \mathtt{pub}} \mathtt{drinks}[y]\right)$ (drinker paradox)

☒ $\exists_{x \in \mathtt{pub}} \neg \mathtt{drinks}[x]$ (A#1)

□ $\exists_{y \in \mathtt{pub}} \mathtt{True}$ (someone in pub)

Do an inference step.

□ Proof Situation

□ $\exists_{x \in \mathtt{pub}} \left(\mathtt{drinks}[x] \Rightarrow \forall_{y \in \mathtt{pub}} \mathtt{drinks}[y]\right)$ (drinker paradox)

□ $X \in \mathtt{pub}$ (A#2)

□ $\neg \mathtt{drinks}[X]$ (A#3)

□ $\exists_{y \in \mathtt{pub}} \mathtt{True}$ (someone in pub)

Fig. 1. Two dialog windows arising in the interactive proof of the drinker paradox.

The interactive dialogs display the current proof situation in the bottom part of the respective windows, on light-brown background. In each case, the top-most formula (above the black line) is the current goal, whereas the formulas below the black line are the assumptions. The check-boxes next to the formulas indicate whether the respective formulas have been activated or deactivated by the user. Above the proof situation, the name of inference rule to be applied next, as chosen by the user, is displayed; it is applied by simply hitting the Enter-key.

3 Higher-Order Rewriting

Rewriting constitutes one of the core components of theorem proving in Theorema as well as many other proof assistants: assumptions (equalities, equivalences, implications) are transformed into rewrite-rules which may then be used to rewrite other formulas in the current proof situation. By default, Theorema can only deal with *first-order* formulas and rewrite-rules, respectively, in the sense that the left-hand-side of a rewrite-rule has to match an expression syntactically in order to be applicable; no $\alpha\beta\eta$-equivalence is taken into account. However, a lot of formulas one frequently encounters in mathematical theories are not first- but higher-order (typical examples are induction rules), and should be treated as such for efficiently working in the respective theories in Theorema.

The higher-order rewriting mechanism we describe in this section serves exactly said purpose: it is able to translate (potentially higher-order) rewrite-rules $\rho : l \mapsto r$ originating from Theorema formulas into *Mathematica* transformation rules `p :> b` that can later be applied by simply calling the standard rule-application-functions from *Mathematica*'s algorithm library (`ReplaceAll`, `ReplaceList`, etc.), such that the correctness condition

$$e' \in \texttt{ReplaceList[}\,e,\ \texttt{p :> b]} \;\Rightarrow\; e \rightarrow_\rho e' \tag{1}$$

is met (where $e \rightarrow_\rho e'$ means that expression e can be rewritten into e' by rule ρ modulo $\alpha\beta\eta$-equivalence). The other direction of (1), though desirable in principle, is out of reach in general if ρ is a higher-order rule: whether higher-order matching (which is one of the key ingredients of rewriting) is decidable or not is still an open problem.[3] Hence, if ρ is higher-order, the *Mathematica* transformation rule `p :> b` cannot be expected to fully reflect the higher-order nature of ρ *in any case.*

3.1 Main Idea

Sometimes, the strategy to tackle problems related to (possibly) undecidable, infinitary matching in concrete implementations is to restrict the class of left-hand-sides of rewrite-rules to so-called *higher-order patterns*; for instance, the simplifier in the Isabelle proof assistant by default can only handle rules falling into this category (see [7], pp. 205–206). The main idea behind our mechanism is similar but less restrictive: if the compiler (i. e. the function that turns rewrite-rules into *Mathematica* transformation rules) can infer that the matching problem associated to the left-hand-side of a given rule ρ is *unitary*, because bound variables appearing among the arguments of free higher-order variables can be used to uniquely determine the instances of these variables when matching an expression, then ρ is "accepted" and turned into a transformation rule that *exactly* corresponds to ρ. Otherwise, if the compiler cannot infer that the matching problem is unitary (maybe because it simply is not), some free higher-order variables have to be treated just like first-order variables that need to match syntactically, meaning that the resulting transformation rule does not correspond to ρ exactly.

Example 1. The left-hand-side of the rewrite-rule (with P, T and S being free variables)

$$\forall_{i=1,\ldots,|T|+|S|}\ P(\texttt{join}(T,S)_i) \mapsto \forall_{i=1,\ldots,|T|}\ P(T_i) \wedge \forall_{i=1,\ldots,|S|}\ P(S_i)$$

is no higher-order pattern, but can still be handled without much ado by our mechanism because the occurrence of the bound variable i in the argument of

[3] Higher-order matching is known to be decidable under certain restrictions on the types involved [6], as well as for general problems below order five [4].

P on the left-hand-side uniquely determines the instance of P when matched against a concrete expression. In contrast, the free variable P in the (nonsense) rule

$$P(0) \mapsto \exists_x P(x)$$

must be treated like a first-order variable by the compiler, for otherwise the instance of P would not be unique in general: matching $0 < 1$ could be accomplished by instantiating P either by $\lambda_x\ 0 < 1$ or by $\lambda_x\ x < 1$, leading to fundamentally different instances of the right-hand-side.

3.2 Implementation Details

The compiler translates rewrite-rules into ordinary *Mathematica* transformation rules. Hence, since *Mathematica* only supports syntactic matching, all possible higher-order aspects ($\alpha\beta\eta$-equivalence, automatic instantiation by λ-terms, etc.) have to be encoded explicitly in the pattern `p` and the body `b` of the resulting transformation rules, e. g. by means of *Mathematica*'s `Condition` function.

Example 2. Consider the higher-order rewrite-rule

$$\sum_{i=1,\ldots,n+1} F(i) - \sum_{i=1,\ldots,n} F(i) \mapsto F(n+1)$$

Ignoring syntactical details of the internal representation of Theorema expressions and technicalities related to capture-avoiding substitutions, the *Mathematica* transformation rule automatically generated by the compiler reads as something like

```
Sum[{i1_, 1, n_+1}, F1_] - Sum[{i2_, 1, n_}, F2_] :>
    substFree[F1, {i1 -> n + 1}] /;
        alphaEquiv[substFree[F1, {i1 -> i2}], F2]
```

As can be seen, higher-order variables (like F in the previous example) are actually never instantiated by concrete λ-terms, but rather the instances of the right-hand-sides of rules are constructed by directly replacing certain sub-expressions (like the bound variable i above). This saves expensive (capture-avoiding) substitutions and explicit β-reductions and, for that reason, is a general principle the compiler adheres to. Moreover, apparently there is no hard-coded, general-purpose higher-order matching algorithm that is attached to every transformation rule, but rather every single transformation rule is equipped with its very own, tailor-made, dynamically generated, optimized algorithm that does not perform any redundant operations. In the example above, only the α-equivalence of two expressions has to be checked in addition to the default syntactic matching carried out by *Mathematica*—a fact the compiler detects and exploits fully automatically when generating the transformation rule.

3.3 More Features

Due to the lack of space, the preceding sections could only provide a glimpse of the higher-order rewriting mechanism, and in particular of the transformation rule compiler; more detailed information can be found in our forthcoming PhD thesis [2]. Still, we want to briefly mention two further features also here:

- Conditional rules, n-ary higher-order variables, and sequence variables are supported as well (to a certain extent). Conditional rules do not cause any difficulties at all, but the presence of free higher-order variables with arity > 1 or free sequence variables complicates matters considerably.
- The compiler by default applies a range of optimizations to the rules it generates for increasing efficiency.
- The condition on matching problems associated to the left-hand-sides of rules being unitary can be relaxed in some situations.

4 Theory Analysis

The third an last tool presented in this paper, called TheoryAnalyzer, enables the automatic analysis of the logical structure of one or several Theorema-theories (i. e. content notebooks together with external proof files). The main idea behind the TheoryAnalyzer is simple enough: read the proof files, and from each proof file store the proof goal and the list of assumptions as the nodes of a graph G that eventually reflects the dependencies between all the formulas thus collected. Namely, a formula φ depends on another formula ψ iff ψ is used as an assumption in a proof of φ; in such a case, G contains a directed edge from ψ to φ.

Once G has been constructed, it can easily be analyzed by means of well-known graph-theoretic functions (like exhaustive search); in particular, it is possible to

- inspect all direct/indirect assumptions/consequences of a given node (corresponding to a formula in the theory),
- detect cycles in the graph, corresponding to circular arguments in the theory,
- find the logical relation between two nodes/formulas, and
- visualize theory dependency graphs and formula statistics diagrams (the latter display the numbers of formulas in each theory); see Fig. 2.

The development of the TheoryAnalyzer was mainly triggered by the practical experience we gained from formalizing Gröbner bases theory in Theorema: it turned out that quite frequently it becomes necessary to re-structure existing parts of formalizations, e. g. by slightly modifying formulas that have already been used as assumptions in proofs. In such situations, the responsibility for maintaining the coherence the formalization exclusively is with the user of Theorema; the system itself does not automatically initiate the re-proving of existing theorems affected by changes in the background theory. Therefore, knowing which theorems *are* affected is of utmost importance—and this is exactly where the TheoryAnalyzer comes into play.

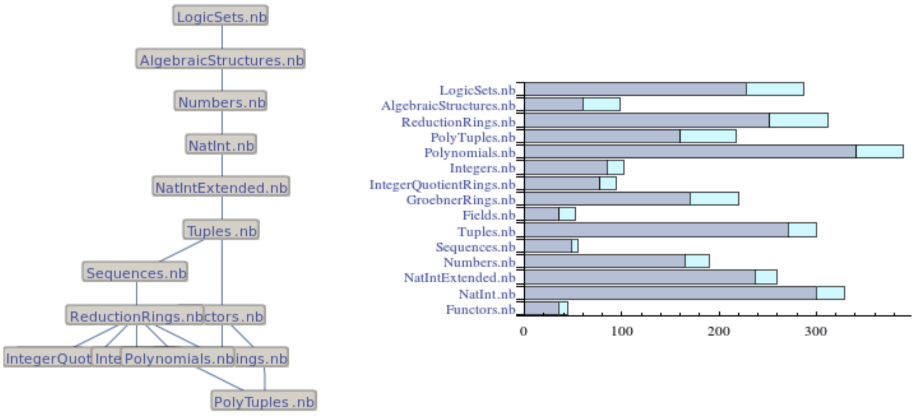


Fig. 2. The theory dependency graph and formula statistics diagram of the Gröbner bases formalization, as automatically generated by the TheoryAnalyzer.

5 Conclusion

In the preceding sections we gave account on three new tools for Theorema 2.0 that already proved extremely useful in practice and are expected to be integrated into the official version of the system in the near future.

There are several directions for further improving the tools: for instance, the dialog-oriented interactive proof strategy could be enhanced by a text-based interface, making it more attractive for people used to such interfaces.

References

1. Buchberger, B., Jebelean, T., Kutsia, T., Maletzky, A., Windsteiger, W.: Theorema 2.0: Computer-Assisted Natural-Style Mathematics. Journal of Formalized Reasoning 9(1), 149–185 (2016)
2. Maletzky, A.: Computer-Assisted Exploration of Gröbner Bases Theory in Theorema. Ph.D. thesis, Research Institute for Symbolic Computation, Johannes Kepler University Linz, Austria (2016), to appear
3. Maletzky, A.: Mathematical Theory Exploration in Theorema: Reduction Rings (2016), submitted to CICM'2016, preprint on `http://arxiv.org/abs/1602.04339`
4. Padovani, V.: Filtrage d'ordre supérieure. Ph.D. thesis, Université Paris 7, Paris, France (1996)
5. Piroi, F., Kutsia, T.: The Theorema Environment for Interactive Proof Development. In: Sutcliffe, G., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning (Proceedings of LPAR'05, Montego Bay, Jamaica, December 2–6). Lecture Notes in Artificial Intelligence, vol. 3835, pp. 261–275. Springer (2005)
6. Stirling, C.: Decidability of Higher-Order Matching. Logical Methods in Computer Science 5(3), 1–52 (2009)
7. Wenzel, M.: The Isabelle/Isar Reference Manual (2016), part of the Isabelle documentation (`https://isabelle.in.tum.de/documentation.html`)