RISC
RESEARCH INSTITUTE FOR
SYMBOLIC COMPUTATION

JKU
JOHANNES KEPLER
UNIVERSITY LINZ

Gray-Box Proving in Theorema

W. Windsteiger

July 2024

RISC Report Series No. 24-07

ISSN: 2791-4267 (online)

Available at https://doi.org/10.35011/risc.24-07

JOHANNES KEPLER
UNIVERSITY LINZ
Altenberger Str. 69
4040 Linz, Austria
www.jku.at
DVR 0093696

Gray-Box Proving in Theorema

Wolfgang Windsteiger
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University Linz (JKU)
Altenbergerstraße 69, 4040 Linz, Austria
Wolfgang.Windsteiger@risc.jku.at, ORCID-ID 0000-0002-7449-8388

***Abstract*—Many theorems in mathematics have the form of an implication, an equivalence, or an equality, and in the standard prover in the Theorema system such formulas are handled by rewriting. Definitions of new function- or predicate symbols are yet another example of formulas that require rewriting in their treatment in the Theorema system. Both theorems and definitions in practice often carry conditions under which they are valid. Rewriting is, thus, only valid in cases where all side-conditions are met. On the other hand, many of these side-conditions are trivial and when presenting a proof we do not want to distract the reader with lengthy derivations that justify the side-conditions. The goal of this paper is to present the design and implementation of a mechanism that efficiently checks side-conditions in rewriting while preserving the readability and the explanatory power of a mathematical proof, which has always been of central interest in the development of the Theorema system.**

***Index Terms*—Theorema, Automated Theorem Proving, Conditional Rewriting, Mathematica, Wolfram Language**

I. Introduction

For the key ideas behind the Theorema system we refer to the survey papers [1]–[3], the only important aspects for understanding the motivation behind this work are the following:

- Theorema aims to support mathematicians and mimics the language used by mathematicians.
- Theorema contains an automated theorem prover.
- Theorema generates proofs in natural mathematical language.
- The trust in automated proofs generated by Theorema is based on the fact that the proofs can easily be read, understood, and checked by humans, because the style of proofs comes close to how mathematicians write up proofs and, in addition, every logical step is spelled out.
- The Theorema system is a multi-method system, its standard prover is based on a natural-deduction-like calculus.
- Theorema is implemented as a Mathematica package using the native Mathematica programming language, nowadays called Wolfram Language.

The standard style of provers propagated in the Theorema system are "white-box-provers" in the sense that they explain every single logical step they apply. The challenge for these provers is that they need to *generate* proofs in such a way that they appear natural when presented as they are. An alternative to this would be *proof transformation*, where, in a post-processing step, a full proof is transformed in such a way that its presentation appears in natural style. The philosophy of Theorema, however, is to investigate *how* mathematicians generate the proofs that they consider elegant and develop automated methods that produce a similar result.

Due to its imitation of human-style proving, the Theorema system is very well-suited for being used in an educational context. We currenty use it in a logic course where teaching how to do "real mathematical proofs" in a style like mathematicians do them is one of the goals of the course. We therefore want to illustrate our idea of "gray-box proving" with a simple example taken from that course. Suppose we had to prove that '1' is the smallest element of the natural numbers with respect to the divisibility relation given the definition of what *smallest* means.

Definition: Let p be a partial order on A and $s \in A$. We call s the *smallest element* in A w.r.t. p iff $p[s, x]$ for all $x \in A$.

Written in Theorema language this definition says nothing else than

$$\forall_{\substack{s,p,A \\ po[p,A] \wedge s \in A}} smallest[s, A, p] :\Longleftrightarrow \forall_{x \in A} p[s, x] \qquad (1)$$

If we need to prove $smallest[1, \mathbb{N}, div]$, the standard Theorema white-box prover would create 3 subgoals:

$$po[div, \mathbb{N}] \qquad 1 \in \mathbb{N} \qquad \forall_{x \in \mathbb{N}} div[1, x]$$

In the presentation of the proof this step would then appear like this:

> In order to prove $smallest[1, \mathbb{N}, div]$, by definition (1), we have to show:
> 1) $po[div, \mathbb{N}]$: ...
> 2) $1 \in \mathbb{N}$: ...
> 3) $\forall_{x \in \mathbb{N}} div[1, x]$: ...

In each of the three branches the "..." indicate that a detailed explanation of the logical reasoning will be given for the respective goal. Our aim is to devise methods that result in a step like

> In order to prove $smallest[1, \mathbb{N}, div]$, *due to* ..., we have to show $\forall_{x \in \mathbb{N}} div[1, x]$,

where the "..." are just references to formulas, from which it "follows easily" that the other conditions are fulfilled. Essentially, we use the definition to rewrite left-hand side to right-hand side and put the side-conditions into a brief hint. This is how mathematicians would probably write this step. However, it is important that an automated reasoner must

not simply omit the justification steps (like mathematicians sometimes do), it just concerns presentation.

This paper explains a mechanism that efficiently proves simple statements and explains from which knowledge the statements' truth can be derived. In Section II we introduce the basic mechanism for condition-checking and describe its implementation within Theorema 2.0. In Section III we investigate other situations, apart from unfolding definitions as described above, that are handled by rewriting in Theorema 2.0 and discuss the role of checking the side-conditions there. Finally, Section IV gives a full example and Section V gives some ideas on future directions related to conditional rewriting in the Theorema system.

II. Efficient Checking of Side-Conditions

In this section we propose an *efficient mechanism* that allows to prove statements that can be *easily derived* from a given knowledge base. The intended use of this mechanism within the Theorema system is in places where we need to

- quickly verify the truth of simple statements, such as atomic formulas without quantifiers, and
- only need rough informations about the logical reasoning behind the scenes.

With "rough informations" we mean that the envisaged mechanism should not be just a black box that simply delivers true or false without further reasons, nor should it be as verbose as the standard prover in Theorema that essentially spells out every logical step.

Example. As a guideline for what we are aiming at, let us consider the following concrete proof situation: suppose the current knowledge base contains at least

1) f is bijective from X to Y iff f is injective and surjective from X to Y,
2) if f is invertible from X to Y then f is bijective from X to Y, and
3) $\exp$ is invertible from $\mathbb{R}$ to $\mathbb{R}^+$,

and that, in order to apply a certain transformation in a proof, we need to verify that the exponential function is indeed surjective from $\mathbb{R}$ to $\mathbb{R}^+$. The result should be something like $\{\text{True}, \{1, 2, 3\}\}$ so that we are able to generate explanatory text like "the transformation ... can be applied due to (1), (2), and (3)". Neither should the transformation be applied silently nor should the justification consume half a page of the proof.

More formally, we consider a statement U *easily derivable* (from K) if

- $U \in K$ or
- $(\underset{x}{\forall} S \Rightarrow T) \in K$ with a substitution σ s.t. $T\sigma = U$ and $S\sigma$ is easily derivable.

In other words, a formula is easily derivable if it is contained in the knowledge base or it matches the right-hand side of an implication in the knowledge base so that it can be derived by generalized universal Modus Ponens (see also Section III-A) from another easily derivable formula, i.e. it goes back to formulas in the knowledge base by *backward chaining* (see also Section III-B). It is obvious how a function `quickCheck`, which checks for easy derivability, can be implemented as a recursive procedure using pattern matching in Mathematica[1]. The function `quickCheck` takes 3 arguments, the formula to be checked, the knowledge base, and a list of formulas already used in the derivation so far. It returns a list consisting of a boolean value indicating whether the formula is derivable from the knowledge and a list of formulas needed in the entire derivation. The base cases would be

$$\texttt{quickCheck}[\texttt{f\_}, \{\texttt{\_\_\_}, \texttt{f\_}, \texttt{\_\_\_}\}, \texttt{U\_}] := \{\texttt{True}, \texttt{Union}[\texttt{U}, \{\texttt{f}\}]\}$$
$$\texttt{quickCheck}[\texttt{f\_}, \texttt{K\_}, \texttt{U\_}] := \{\texttt{False}, \{\}\}$$

and every (quantified[2]) implication $f \equiv \underset{x}{\forall} S \Rightarrow T$ in the knowledge base would dynamically add a recursive case[3]

$$\texttt{quickCheck}[\texttt{T}^*, \texttt{K\_}, \texttt{U\_}] := \texttt{quickCheck}[\texttt{S}, \texttt{K}, \texttt{Union}[\texttt{U}, \{\texttt{f}\}]]$$

Pattern matching available in the Mathematica programming language allows to implement the different recursion cases as individual functions differing in the input patterns. This allows for easy "extension" of the program through adding additional cases. At this point we want to dive a little bit deeper into implementation issues. If one would implement the recursion as suggested above it would at run-time take the first matching implication for backchaining and continue the recursion. If successful then everything is fine, but if not then we need to *backtrack* and try another reduction if possible. In the above implementation, Mathematica would not backtrack but simply return $\{\texttt{False}, \{\}\}$.

The solution for this is to implement recursion differently, namely by attaching a condition to each case and do the recursive call from inside the condition. Roughly, the implementation pattern for this is

$$\texttt{quickCheck}[\texttt{T}^*, \texttt{K\_}, \texttt{U\_}] := \tag{2}$$
$$\texttt{Module}[\texttt{v}, \texttt{body /; } \ldots \texttt{qCQ}[\texttt{S}, \texttt{K}, \texttt{U}] \ldots]$$

In the Wolfram Language, $\texttt{Module}[v, body\ /;\ cond]$ is a construct that executes $body$ if $cond$ evaluates to `True`, where $body$ and $cond$ may share local variables listed in v. `qCQ` is just a wrapper around `quickCheck` that, instead of returning $\{b, U\}$, returns only the boolean value b and stores the used formulas U in a global variable, from which they can be retrieved later. This allows the quickCheck-mechanism to be used inside boolean conditions directly. The dots around `qCQ` should indicate that the condition may consist of more than just the quickCheck. In fact, it can be any sequence of Mathematica commands, whose last value will be taken as the value of the entire condition. In particular, during the evaluation of the condition one may assign values to local variables from v that can then be used during the evaluation of $body$, which is useful

[1] We only sketch the implementation in order to give an impression, we omit all details concerning, e.g., formula data-structures in Theorema.

[2] The variable x under the quantifier actually denotes the set of all quantified variables

[3] $\texttt{T}^*$ is a symbolic way of writing "the expression T with all occurrences of the variable x replaced by the pattern x_".

for preventing multiple execution of expensive operations in both the condition and the body.

The method just presented is applied to quantified implications whose left-hand *and* right-hand sides are atomic formulas. If either side is a quantifier formula, it is neglected in the backchaining process. For other formats of formulas in the knowledge base we proceed as follows:

- If $f \equiv \underset{x}{\forall}(S \Rightarrow T_1 \wedge \ldots \wedge T_n)$: Since f is equivalent to the conjunction of the individual $\underset{x}{\forall}(S \Rightarrow T_i)$ we generate individual quickCheck-cases for each T_i.
- If $f \equiv \underset{x}{\forall}(S_1 \vee \ldots \vee S_n \Rightarrow T)$: Since f is equivalent to the conjunction of the individual $\underset{x}{\forall}(S_i \Rightarrow T)$ we generate individual quickCheck-cases for each S_i.
- If $f \equiv \underset{x}{\forall}(S_1 \wedge \ldots \wedge S_n \Rightarrow T)$ then backchaining must branch to all the S_i and it delivers `True` only if all individual branches succeed. In the implementation this is reflected by calling `quickCheck` with a list of formulas as first parameter.
- If $f \equiv \underset{x}{\forall}(S \Leftrightarrow T_1 \wedge \ldots \wedge T_n)$ with atomic S then it is processed as if it was an implication. For the dual case where T is atomic and S is a conjunction we proceed analogously.
- If $f \equiv S :\Leftrightarrow T_1 \wedge \ldots \wedge T_n$ then we treat it like an implication. Note that in this case S is always atomic.

The treatment of definitions in the last case has a nice effect. In mathematical practice there are many definitions of this special form, just think of *a function is bijective iff it is injective and surjective* or *G is a group iff it is a monoid and every element is invertible*. If we then have S in the knowledge base and need to show (check/verify) one of the T_i we would normally have to wait until the prover expands the definition of S in order to get T_i into the knowledge base. On the other hand, as a general strategy, we prefer to expand definitions rather late in the proof search because expanding definitions early tends to mess up expressions. This effect is amplified when we work with definitions of some level of nesting, e.g., group – monoid – semigroup – groupoid. With our quickCheck-mechanism there is no need to wait for the unfolding of the definition because, e.g., if we know G is a group and want to check whether it is a groupoid, the process would backchain along the definitions from groupoid to semigroup to monoid to group and finally find that G is a group in the knowledge base without unfolding a single definition.

Finally, we want to describe how this mechanism deals with cases where backchaining leads from T to S and S has free variables that *do not occur* in T. Essentially, when a backchaining step introduces free variables they need to be thought of as existentially quantified, more logical details will be given in Section III-B, where we describe one single backchaining step as a reasoning rule in the Theorema standard prover. In the Theorema implementation backchaining from T to S via some quantified implication happens through a recursive call as shown in (2). When a new formula goes to the knowledge base during a proof this Mathematica "program" needs to be generated automatically by translating Theorema syntax to Mathematica programming language. In this step the free variables of T are identified and are turned into patterns in order to get T^* on the left-hand side whereas they are converted to the respective symbols without pattern notation in the right-hand side S. Additional free variables in S can easily be identified because they stay unconverted and are, thus, still Theorema free variables that can be recognized on the datastructure-level.

Extra care has to be taken in connection with branching during backchaining, since branching on conjunctions (see above) should be avoided as soon as free variables are present. However, *not being able at all* to cope with these cases would make the whole mechanism nearly useless, since backchaining over two to three levels *almost always*[4] introduces free variables. On the other hand, for an efficient quickCheck-mechanism for easy derivability it would not be appropriate to now employ a full-power existence prover. What we do instead is again a simple heuristic that still often succeeds. Suppose we have $S := \{S_1, \ldots, S_n\}$ to be verified with free variables, then we choose (if possible) i s.t. S_i contains all free variable and then we call `quickCheck` recursively on S_i passing $S \setminus \{S_i\}$ as an additional fourth parameter. `quickCheck` with four parameters then tries to instantiate the free variables in the first formula by finding a matching formula in the knowledge base and, if successful, instantiates all formulas given as fourth parameter with the same substitution and continues recursively with the instantiated set.

Even this simple mechanism can turn out cumbersome in practical examples. First it may backtrack when choosing a formula containing all free variables and then it may backtrack again when choosing a matching formula from the knowledge base. This process may, of course, fail since in some cases only step-by-step partial instantiations can be done, but this would soon become tricky—we therefore consider our approach a reasonable compromise.

This mechanism turned out extremely useful and powerful, we therefore not only use it for checking side-conditions, which was the original motivation as described in Section I. We use `quickCheck` now also when checking proof termination. Prior to that a proof only succeeded when the goal was contained in the knowledge base K or it was an instance of a universally quantified formula in K. Now it succeeds already when the goal "follows easily" from K. Furthermore, we employ `quickCheck` when we face an existential goal, since the instantiation mechanism described above proves very useful in these cases.

III. Conditional Rewriting in Theorema 2.0

Transformation rules with patterns and conditions and the application of rules to other expressions are a power-

[4]We have not done enough and systematic tests to give precise numbers, but it was felt that quickCheck could often *not* verify simple facts due to this reason.

ful tool in Mathematica. Rules with conditions are written[5] `S[x_] /; C :> T`. The application of such transformation rules to various types of expressions can be considered, essentially, as *conditional rewriting*. The big drawback of this approach is that the condition C is *evaluated by computation* in the Mathematica kernel when we apply a transformation rule like the above. In the context of proving, what we actually need is to *prove* C w.r.t. the current knowledge base, and, if successful, deliver an explanation to the user *why* C is true. However, we consider typical side-conditions to be "easily provable", therefore we are not aiming at a full-fledged prover but rather use our quick-check mechanism presented in Section II. In the rest of this section we collect scenarios, where conditional rewriting can effectively applied in our standard white-box reasoners.

A. Knowledge Expansion

With "knowledge expansion" we mean the process of inferring new knowledge from already known facts. A very basic form of knowledge expansion happens when we apply the Modus Ponens inference rule, which says that when we have both $S \Rightarrow T$ and S in our knowledge base we can join T to the knowledge base, i.e., we can *infer new knowledge* T. A very efficient way of implementing this in Mathematica is to convert $S \Rightarrow T$ in to a transformation rule `S :> T` and apply it to all formulas in the knowledge base. If S occurs in the knowledge base then the rule will turn it into T, all other formulas will be left unchanged because the rule would not match. Hence, we can add all formulas that changed, i.e., T, to the knowledge base.

The generalized universal Modus Ponens[6] says that when we have both $\underset{x}{\forall}(S \Rightarrow T)$ and *an instance of* S in our knowledge base we can join the *respective instance* of T to the knowledge base. Adopting the technique shown above, this can easily be achieved by turning the universally quantified implication into transformation rule `S* :> T` and apply it to all formulas in the knowledge base. The pattern `S*` will then apply to *all instances* of S and the rule will produce the *respective instance* of T to be joined to the knowledge base.

Simple Example. The quantified implication $\underset{x}{\forall}(4|x \Rightarrow even(x))$ would be represented by the rule `4|x_ :> even[x]`. Suppose we have $4|20$ in our knowledge base, then, since $4|x\_$ matches $4|20$, the rule applies to $4|20$ and results in $even(20)$, which corresponds to inferring $even(20)$ from $4|20$ using the quantified implication.

Even more generally, consider now implications of the form

$$\underset{x}{\forall}(S_1 \wedge S_2 \wedge \cdots \wedge S_n \Rightarrow T), \tag{3}$$

which we, in fact, encounter very frequently in mathematics, since many theorems actually exhibit this format. Such an implication is, of course, equivalent to a "nested implication"

$$\underset{x}{\forall}\big(S_1 \Rightarrow (S_2 \Rightarrow \ldots \Rightarrow S_{n-1} \Rightarrow (S_n \Rightarrow T))\big),$$

and applying generalized universal Modus Ponens would then result in a cumbersome step-by-step inference from knowing S_1 to an implication of depth one less, then from knowing S_2 to an even shallower one, etc. until finally deriving T from S_n. However, for any choice $1 \leq i \leq n$, another alternative equivalent formulation is

$$\underset{x}{\forall}\big((S_1 \wedge \cdots \wedge S_{i-1} \wedge S_{i+1} \wedge \cdots \wedge S_n) \Rightarrow (S_i \Rightarrow T)\big),$$

which could be interpreted intuitively as "from an instance of S_i we can infer the respective instance of T provided we know also respective instances of $S_1, \ldots, S_{i-1}, S_{i+1}, \ldots, S_n$". With conditional rewriting in mind, the rule[7]

$$\mathtt{S}_i^* \,/;\, \mathtt{qCQ}[\{\mathtt{S}_1, \ldots, \mathtt{S}_{i-1}, \mathtt{S}_{i+1}, \ldots, \mathtt{S}_n\}] :> \mathtt{T}$$

would achieve exactly what we need. The left-hand side of the rule S_i should contain all variables bound by the universal quantifier or at least all variables occurring in the right-hand side T in order to guarantee that during the pattern matching process all variables get instantiated and no free variables remain in T[8]. Hence, we generate a transformation rule for every i such that S_i is an atomic formula with no free variables other than the free variables in T.

B. Goal Reduction

The goal-oriented application of Modus Ponens is usually referred to as backward chaining and uses an implication $S \Rightarrow T$ to replace the proof goal T by S. In a more general form it uses quantified implications of the form (3) for replacing the current goal T by subgoals $S_1, \ldots, S_n$. This is exactly the mechanism used in our quick-check procedure described in Section II. In white-box proving, however, we only apply one step at the time instead of recursively continuing including backtracking.

In practice, T does often not contain some of the universally quantified variables x. Suppose those variables are called y whereas z are the remaining variables in x, and let furthermore $C = \{C_1, \ldots, C_k\}$ and $S' = \{S'_1, \ldots, S'_m\}$ be such that $\{S_1, \ldots, S_n\} = C \cup S'$ and every C_j is free of y and every S'_j contains y. Then (3) is equivalent to

$$\underset{z}{\forall}\big((\underset{y}{\exists} S_1 \wedge \cdots \wedge S_n) \Rightarrow T\big).$$

which is in turn equivalent to

$$\underset{z}{\forall}\big((C_1 \wedge \cdots \wedge C_k) \Rightarrow (\underset{y}{\exists} S'_1 \wedge \cdots \wedge S'_m) \Rightarrow T\big).$$

[5]The underscore in "$x\_$" is part of the pattern language of Mathematica and it means that x can take any value.

[6]Note that this is not a new inference rule of logic, it is just a combination of universal instantiation and classical Modus Ponens.

[7]`qCQ`$[\{\mathtt{S}_1, \ldots, \mathtt{S}_n\}]$ means to prove all the $\mathtt{S}_j$ simultaneously.

[8]"Free" variables in T would need to be bound by a universal quantifier but we deliberately decide not to introduce new quantifiers when inferring new knowledge.

Hence, for every formula in the knowledge base shaped like (3), where T is an atomic formula, we generate a transformation rule

$$\mathtt{T}^* \,/;\, \mathtt{qCQ}[\{\mathtt{C}_1, \ldots, \mathtt{C}_k\}] \;:>\; \underset{y}{\exists}\, \mathtt{S}'_1 \wedge \cdots \wedge \mathtt{S}'_m,$$

If $S' = \emptyset$, i.e., no subgoal contains free variables in addition to those appearing in T, then the right-hand side simplifies to `True`[9].

Example. The quantified implication

$$\underset{f,X,Y,B}{\forall} \big((f\colon X \to Y \wedge B \subseteq Y \wedge \mathcal{I}(X,f) = B) \Rightarrow surjective(f,X,B)\big)$$

would lead to the rule

$$\mathtt{surjective}[\mathtt{f\_}, \mathtt{X\_}, \mathtt{B\_}]\ /;\ \mathtt{qCQ}[\mathtt{I}(\mathtt{X},\mathtt{f}) = \mathtt{B}]\ :> \underset{\mathtt{Y}}{\exists}(\mathtt{f}\colon \mathtt{X} \to \mathtt{Y} \wedge \mathtt{B} \subseteq \mathtt{Y})$$

With this rule the proof of, e.g., the surjectivity of $f(x) := x^2$ from $\mathbb{R}$ to $\mathbb{R}_0^+$ can be reduced to finding a Y such that $f\colon \mathbb{R} \to Y$ and $\mathbb{R}_0^+ \subseteq Y$ provided that we can "easily show" that $\mathcal{I}(\mathbb{R}, f) = \mathbb{R}_0^+$. This would be the case, e.g., if $\mathcal{I}(\mathbb{R}, f) = \mathbb{R}_0^+$ is in the knowledge base, i.e., goal reduction would wait until this knowledge is available.

Example. A variant from the example above would be

$$\underset{f,X,Y}{\forall} \big((f\colon X \to Y \wedge \mathcal{I}(X,f) = Y) \Rightarrow surjective(f,X,Y)\big),$$

which gives the rule

$$\mathtt{surjective}[\mathtt{f\_}, \mathtt{X\_}, \mathtt{Y\_}]\ /;\ \mathtt{qCQ}[\{\mathtt{f}\colon \mathtt{X} \to \mathtt{Y}, \mathtt{I}(\mathtt{X},\mathtt{f}) = \mathtt{Y}\}]\ :>\ \mathtt{True}$$

With this rule the proof of, e.g., the surjectivity of $\exp$ from $\mathbb{R}$ to $\mathbb{R}^+$ would immediately succeed as soon as we know $\exp\colon \mathbb{R} \to \mathbb{R}^+$ and $\mathcal{I}(\mathbb{R}, \exp) = \mathbb{R}^+$.

C. Explicit Definitions, Equalities, and Equivalences

The handling of definitions has often been a source of dispute when discussing the logical formal foundations of automated provers or proof assistants. While (ZF) set theory and (first-order) predicate logic is widely considered to be the mathematicians' logical foundation, many mechanized systems prefer variants of type theory as their logical foundation. In set theory, "everything is a set" so the language itself lacks any distinction between "different sorts" of objects. The problem with most variants of predicate logic is, in a nutshell, that conservative language extensions by adding new function and predicate symbols "on the fly" through definitions assumes the functions and predicates as *total*, i.e., they are defined for all objects in the domain of discourse. As a remedy, foundations that support partial functions in one or the other way have been proposed as a logical basis for mechanized mathematics, see [4]–[6]. We shall not discuss details about different foundations, nor do we want to suggest new logical foundations. Instead, we want to present the pragmatic approach taken in the Theorema system.

One of the standard ways for introducing new function or predicate constants in predicate logic is via (explicit) definitions, see, e.g., [7]. There are various notations around, in mathematical texts they often appear like

$$f(x) := T \quad \text{or} \quad p(x) :\Leftrightarrow R,$$

where f and p are *new* function (predicate) constants and T (respectively R) are terms (respectively formulas). The convention is that these notations are just abbreviations for the new constants' *defining axioms*

$$\underset{x}{\forall}(f(x) = T) \quad \text{and} \quad \underset{x}{\forall}(p(x) \Leftrightarrow R),$$

which are added to the knowledge base through these definitions. In mathematical practice, definitions are (almost) always augmented with restricting conditions on the variables. The motivation for attaching additional conditions is on the one hand to formally avoid "invalid expressions" (like a division by zero) and, on the other hand, much more informally as an indication (for the user) for which "sort of arguments" certain notions are supposed to be applied.

Since the early days, the Theorema syntax for definitions has always supported this style of *conditional definitions*. One reason was that Theorema's user interface was always guided by the style how typical mathematicians would want to write or read mathematics. On top of that, the way how to define new notions in Theorema was certainly influenced also by the style how new concept are defined in the Mathematica system, where we have always considered pattern matching with additional conditions very close to what mathematicians usually do. In practice, mathematicians would not care about "undefined expressions". They would simply not form such expressions or, more importantly, simply "not touch them" in case they have been formed by mistake. In an untyped language like Theorema it is difficult to prevent such expressions from being formed. Therefore, conditions are the perfect mechanism in order to "not touch" expressions that are not formed according to the intentions.

Regardless of the concrete syntax the Theorema system follows the convention that the defining axioms for definitions like

$$\text{let } x \text{ with } C, \text{ then } f(x) := T \quad \text{or}$$
$$\text{let } x \text{ with } C, \text{ then } p(x) :\Leftrightarrow R,$$

are just

$$\underset{x}{\forall}(C \Rightarrow f(x) = T) \quad \text{and} \quad \underset{x}{\forall}(C \Rightarrow (p(x) \Leftrightarrow R)).$$

Through the untyped nature of the Theorema language, the standard Theorema prover can treat quantified[10] equalities and

[9] `True` is a formula constant in the Theorema language. If the proof goal is `True` the proof immediately finishes successfully.

[10] The case of unquantified (atomic) equalities and equivalences is a trivial special case that simply skips instantiation. All considerations apply to this special case with modifications.

equivalences, i.e.,

$$\underset{x}{\forall} S(x) = T \quad \text{and} \quad \underset{x}{\forall}(S(x) \Leftrightarrow T)$$

in the knowledge base identically, namely it tries to replace any occurrence of instances of $S(x)$ by the corresponding instances of T or vice versa[11]. In the cases where the equality or equivalence originates from a definition, the direction is predetermined, it always goes from $S(x)$ to T.

If the subformula T in (3) is an equality $A = B$ or an equivalence $A \Leftrightarrow B$ then we treat such knowledge differently. Instead of inferring new knowledge T or reducing the goal T as described in the previous sections, we use this knowledge to replace A by B or vice versa. The situation is similar to what has been said for conditional definitions in Section I only that we use equalities and equivalences that are not associated to definitions in both directions. For this, we generate twin rewrite rules for both directions

$$\mathtt{A}^* \;/;\; \mathtt{qCQ}[\mathtt{S}_1, \ldots, \mathtt{S}_n] \;:>\; \mathtt{B} \quad \text{and}$$
$$\mathtt{B}^* \;/;\; \mathtt{qCQ}[\mathtt{S}_1, \ldots, \mathtt{S}_n] \;:>\; \mathtt{A}$$

and, in order to avoid cyclic rewriting, we delete a rule as soon as its twin got applied. This approach is different from using path orderings to determine a rewrite rule's orientation classically used in rewriting. In practice, however, it often leads to nicely "natural" proofs and we have so far not run into problems concerning termination of rewriting.

D. Handling of Implicit Definitions

Explicit definitions as discussed in Section I are a special case of definitions, where joining their defining axiom to the current theory is unproblematic as long as the *definiens* (the right-hand side) contains no free variables other than the *definiendum* (the left-hand side). For function constants there is the more general mechanism, we call them *implicit definitions*, where one defines $f(x)$ to be *the* y with a certain property $\phi(x, y)$. Logically, implicit definitions are tightly related to the description operator ι and their sound use depends on the unique existence of a y with $\phi(x, y)$ for every x. In the majority of the cases in practice, unique existence of y only comes *under certain restrictions* for x, which puts conditional rewriting into the focus again.

The Theorema standard prover employs two inference techniques for implicitly defined functions

$$\text{let } x \text{ with } C, \text{ then } f(x) := \text{the } y \text{ such that } \phi(x, y).$$

Firstly, when we need to prove $f(t) = s$ then the definition can be applied if C holds (with x replaced by t) and it is then sufficient to prove $\phi(t, s)$. Secondly, in general, any occurrence of an instance $f(t)$ in the goal or in the knowledge base, for which C holds (with x replaced by t), can be replaced by a new constant τ and knowledge $\phi(t, \tau)$ about the new constant can be added. In both cases, the crucial part in the implementation is realized by turning the definition into a rewrite rule

$$\mathtt{f}[\mathtt{x\_}] \;/;\; \mathtt{qCQ}[\mathtt{C}] \;:>\; \{\mathtt{y}, \phi(\mathtt{x}, \mathtt{y}), \ldots\}.$$

The right-hand side of the rule delivers y separately and leaves y unchanged in ϕ, because y needs to be substituted by different terms in the two scenarios. The "…" indicate that the rule actually delivers some additional information. When applying the rule to a potential instance $f(t)$, the pattern matching of Mathematica takes care about replacing x by t (both in C and ϕ) and checks the validity of C. If applicable, the list delivered contains all information necessary for processing the definition as described above. In particular, it contains the information generated by the prove-mechanism that explains why C holds for t. We do not go into more details, because this would require lengthy implementation details.

IV. Example

In this section we show how all the mechanism presented so far play together in a concrete proof in the Theorema system. We do not show an entire proof but pick out just those steps where the new methods are actually applied. The example is taken from an introductory course for undergraduate mathematicians on the basic proof techniques from logic and it is on basics of functions. As an example for an implicit definition we have the notion of the *pre-image* of a value b under a function f defined as follows: Let f, A, B s.t. $f\colon A \to B$ and $injective(f, A)$, then for all $b \in B$

$$preImage(b, f, A) := \text{the } a \in A \text{ with } f(a) = b. \qquad \text{(pI)}$$

At a certain stage during the proof the knowledge base K contains[12] (among others)

$$\underset{\substack{f, A, B \\ f\colon A \to B}}{\forall} bijective(f, A, B) :\Leftrightarrow$$
$$injective(f, A) \wedge surjective(f, A, B) \qquad \text{(bijective)}$$
$$\underset{f, X, Y, x}{\forall} \big((f\colon X \to Y \wedge x \in X) \Rightarrow f(x) \in Y\big) \qquad \text{(vic)}$$
$$\bar{f}\colon \bar{X} \to \bar{Y} \qquad (2.1)$$
$$bijective(\bar{f}, \bar{X}, \bar{Y}) \qquad (2.2)$$
$$\bar{x} \in \bar{X} \qquad (16)$$

and we need to show

$$preImage(\bar{f}(\bar{x}), \bar{f}, \bar{X}) = \bar{x}. \qquad (15)$$

Note that the goal (15) has exactly the form as discussed in Section III-D, it is an equality with the implicitly defined function $preImage$ on one side. In order to handle this goal, we need to check the side-conditions of definition (pI), i.e.,

$$\{\bar{f}(\bar{x}) \in B, \bar{f}\colon \bar{X} \to B, injective(\bar{f}, \bar{X}), \bar{x} \in \bar{X}\}.$$

[11] Again, we skip details on rewrite ordering and on how to avoid cycles in rewriting.

[12] The over-barred symbols denote constants that have been introduced in previous steps.

The conditions contain a free variable B, hence, the quickCheck-mechanism chooses one of the conditions containing all free variables and tries to find a matching formula in K. This search succeeds with $\bar{f}\colon \bar{X} \to B$ matching (2.1). Thus, B gets instantiated with $\bar{Y}$ and we recursively check $\bar{f}(\bar{x}) \in \bar{Y}$. Now the procedure initiates backchaining based on (vic), which ends up in

$$\{\bar{f}\colon X \to \bar{Y}, \bar{x} \in X\}$$

with free variable X. Similar to above, X will be instantiated with $\bar{X}$ after matching $\bar{f}\colon X \to \bar{Y}$ to (2.1) and the recursive check of $\bar{x} \in \bar{X}$ terminates successfully because this is identical to (16) in K. The process continues with $injective(\bar{f}, \bar{X})$, which reduces by backchaining based on (bijective) to

$$\{\bar{f}\colon \bar{X} \to B, bijective(\bar{f}, \bar{X}, B)\}.$$

Similar to above, B will be instantiated with $\bar{Y}$ after matching $\bar{f}\colon \bar{X} \to B$ to (2.1) and the recursive check of $bijective(\bar{f}, \bar{X}, \bar{Y})$ terminates successfully because of (2.2) in K. Finally, $\bar{x} \in \bar{X}$ remains to be checked, which is true because of (16) in K.

All of the above happens silently in the background, the check of all side-conditions is finally successful and the inference step for the implicitly defined expression as explained in Section III-D is applied. The explanatory text for this step in the final Theorema proof then just reads like

"For proving (15), due to (pI), (bijective), (vic), (2.1), (2.2), and (16), it suffices to prove $\bar{x} \in \bar{X} \wedge \bar{f}(\bar{x}) = \bar{f}(\bar{x})$."

V. Conclusion and Future Work

We have presented an efficient mechanism for checking the truth of simple statements including a brief and compact justification for their truth. We provide an overview over different situations where such a mechanism can fruitfully be applied. The mechanism is implemented in the Theorema system that is used in educational scenarios recently. In particular in teaching how to do correct (and elegant) mathematical proofs it is important that the software that is used to generate "model proofs" does itself produce correct and elegant proofs. Therefore, the efficient checking of side-conditions is an important ingredient for the correctness whereas the compact justification contributes to the proof's elegance.

Concerning related work, the motivation behind "gray-box proving" is similar to that of using "hammers" in interactive theorem proving. Most interactive provers, like Isabelle/HOL, Lean, or Mizar have established mechanism that allow them to quietly close subgoals by connecting to an automated (typically first order) prover or to an SMT solver. A comprehensive overview of these approaches can be found in [8]. However, these approaches are very different from what we describe in this work, because, on the one hand, Theorema is not an interactive system and, on the other hand, applying an ATP or an SMT solver is completely black-box and does not give any indication on how the sobgoals have been proven. Still, both approaches finally end up in considering certain goals as "trivially provable" without showing all details. Theorema does it when the goal can be derived by repeatedly applying Modus Ponens and instantiation, i.e., if backward chaining succeeds, the interactive provers do it when the goal is provable by a fully automated system.

Of course, backward chaining as such is not new and is applied in many automated theorem provers when they work goal-oriented. However, typically this technique is then combined with other inferencing steps such as skolemization, instantiation, or unification. Backward chaining is then usually applied only one step at the time like in Theorema's white-box approach for goal reduction as described in Section III-B in order to allow also other rules to apply. The recursive application including backtracking results in a reasoning scheme reminiscent of how a Prolog interpreter works. Our emphasis concerning the recursive application lies on the Mathematica specific implementation, which gives us backtracking for free without any need to explicitly implementing it. Since Theorema is the only automated theorem prover known to the author implemented in the Mathematica programming language, it is hard to relate that to any other existing implementation.

As for future work, we note that checking whether a formula occurs in the knowledge base, which finally amounts to testing equality of formulas, is an operation that is heavily used during our quick checking. It should, therefore, be investigated whether an enhanced data structure with hash-codes attached to each formula in the knowledge would lead to considerable improvement for checking membership in the knowledge base.

Furthermore, we plan to integrate the presented mechanism also into the computing facilities of Theorema. A standard feature of Theorema since the early days, besides generating automated proofs, is to *compute* values of expressions using definitions of functions and predicates given in the current Theorema session in combination with an executable implementation of the Theorema language. Since this computation is heavily based on conditional rewriting it would be near at hand to check side-conditions during rewriting using the now available quick-check-mechanism described in this work. Compared to just evaluating conditions to true or false the logical power of condition checking would be increased significantly by backchaining combined with searching for instances as described in Sections II and IV.

References

[1] B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger, "Theorema: Towards Computer-Aided Mathematical Theory Exploration," *Journal of Applied Logic*, vol. 4, no. 4, pp. 470–504, 2006.

[2] B. Buchberger, C. Dupre, T. Jebelean, F. Kriftner, K. Nakagawa, D. Vasaru, and W. Windsteiger, "The Theorema Project: A Progress Report," in *Symbolic Computation and Automated Reasoning (Proceedings of CALCULEMUS 2000, Symposium on the Integration of Symbolic Computation and Mechanized Reasoning)*, M. Kerber and M. Kohlhase, Eds., St. Andrews, Scotland. Copyright: A.K. Peters, Natick, Massachusetts, 6-7 August 2000, pp. 98–113.

[3] B. Buchberger, T. Jebelean, T. Kutsia, A. Maletzky, and W. Windsteiger, "Theorema 2.0: Computer-Assisted Natural-Style Mathematics," *JFR*, vol. 9, no. 1, pp. 149–185, 2016. [Online]. Available: http://dx.doi.org/10.6092/issn.1972-5787/4568

[4] W. M. Farmer and J. D. Guttman, "A Set Theory with Support for Partial Functions," *Studia Logica: An International Journal for Symbolic Logic*, vol. 66, no. 1, pp. 59–78, 2000. [Online]. Available: http://www.jstor.org/stable/20016214
[5] M. Beeson, *Foundations of Constructive Mathematics*. Springer-Verlag, Berlin, 1985.
[6] L. G. Monk, "PDLM: A Proof Development Language for Mathematics," MITRE Corporation, Bedford, Massachusetts, Technical report M86-37, 1986.
[7] H. Ebbinghaus, J. Flum, and W. Thomas, *Mathematical Logic*. Springer Verlag New York Berlin Heidelberg Tokyo, 1984, ISBN 0-387-90895-1.
[8] J. C. Blanchette, C. Kaliszyk, L. C. Paulson, and J. Urban, "Hammering towards QED," *J. Formaliz. Reason.*, vol. 9, no. 1, pp. 101–148, 2016. [Online]. Available: https://doi.org/10.6092/issn.1972-5787/4593