A New Reasoning Framework for Theorema 2.0

Alexander Maletzky*

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University Linz, Austria
alexander.maletzky@risc.jku.at

Abstract. We present a new add-on for the Theorema 2.0 proof assistant, consisting of a reasoning framework in the spirit of (though not exactly as) the well-known LCF approach to theorem proving: a small, trusted kernel of basic inferences complemented by an extensive collection of automatic and interactive proof methods that construct proofs solely in terms of the basic inferences. We explain why such an approach is desirable in the first place in Theorema (at least as a possible alternative to the existing paradigm), how it fits together with the current default set-up of the system, and how proof-checking with the inference kernel of the new framework proceeds. Since all this is heavily inspired by the Isabelle proof assistant, we in particular also highlight the differences between Isabelle and our approach.

Keywords: Interactive theorem proving, automated reasoning, Theorema

1 Introduction

The Theorema project aims at creating a software system that supports the 'working mathematician' in his everyday work, be it performing computations, proving conjectures or implementing algorithms—exploring mathematical theories, in short. The resulting Theorema[1] system [2,3], implemented on top of the *Mathematica* computer algebra system,[2] achieves this goal by providing a user-friendly graphical user interface [16], a computation engine seamlessly integrated into a collection of automated and interactive provers, and a quite unique mechanism for automatically turning formal proofs into readable proof documents. Remarkable research done primarily in Theorema includes the development and implementation of a novel method for solving linear boundary value problems [13] and the automated synthesis of Buchberger's Gröbner bases algorithm by lazy thinking [1,5].

Still, there is one aspect Theorema has not paid particular attention to yet: ensuring the logical consistency of formalized theories and the logical correctness of proof methods (mainly those developed by users). So far, the system only

* The research was funded by the Austrian Science Fund (FWF): P 29498-N31

[1] 'Theorema' shall always refer to the current version Theorema 2.0 throughout this paper; http://www.risc.jku.at/research/theorema/software/

[2] http://www.wolfram.com/mathematica/

provides useful tools and functionality for 'doing mathematics' to its users, but it also leaves them with the responsibility to ensure correctness and consistency of formalizations themselves. [3] This might be feasible for small and isolated elaborations, but as soon as they increase in size and rest on existing work, users cannot reasonably take care of these things themselves any more; this, at least, is one of the findings from our recent formalization of reduction ring theory in Theorema [9]. In that sense, the *new reasoning framework*[4] presented in this paper is a first attempt to solve these issues; it is heavily inspired by the Isabelle system [12,11,14]. The main reason for choosing Isabelle among the large variety of existing proof assistants of very different flavor is because Isabelle and Theorema share the ultimate design goal of being flexible systems that do not impose too many constraints on their users regarding the logic they have to work in—be it first-order logic, set theory, higher-order logic, etc. A survey of other approaches and proof assistants in general is given in [15,6].

The rest of this paper is organized as follows: Section 2 outlines the main design principles underlying our new framework. Sections 3 and 4 describe how types and terms are formed, and how the consistency of formal theories can be ensured. Section 5 is the main part of this paper: it describes the logic and inference kernel of the framework. Section 6 sketches how arbitrary object logics can be set up and made available to users. The style of presentation intentionally is on a rather abstract level; only occasionally details of the actual implementation are revealed.

2 Design Principles of the New Framework

As a mathematical assistant system, Theoremahas its strengths and weaknesses.

One apparent strength is its focus on *natural-style* mathematics by supporting common two-dimensional mathematical notation (with bound variables being typeset *under* the respective binders, for instance) both on input and output, and by putting a lot of effort in the presentation of proofs: after a theorem has been proved—no matter whether automatically or interactively—the system can fully automatically generate a nicely formatted *proof document* that explains every step in the proof both formally and informally, in a way human users are able to comprehend even if they do not know Theorema very well.

Furthermore, Theorema traditionally is a very *flexible* system in the sense that it offers its users much freedom in formalizing their mathematics: it does not impose any conditions on how theories must be structured or how new objects must be introduced in order to preserve the logical consistency of the respective theories. Expressions entered into the system are not required to be well-typed, simply because the language of Theorema is completely untyped. And, finally, Theorema encourages its users to develop their own proof methods, tailor-made for the theory they are currently working on, but again without imposing any

[3] Preliminary work on 'lifting' proved facts to the inference level by reflection, summarized in [7], has not been incorporated into the system so far.

[4] `http://www.risc.jku.at/people/amaletzk/AddOns.html`.

conditions whatsoever on how these proof methods must behave in order to keep the reasoning machinery sound. See [2,3] for details.

Our phrasing of the previous paragraph already indicates that these strengths entail weaknesses, too. Great flexibility is convenient for the user who formalizes her mathematics, but it perhaps raises problems for the user who wants to build upon existing formalizations, relying on the correctness of their formal proofs and the logical consistency of the formalized theories. One has to admit that certain doubt with regard to these correctness- and consistency questions remains in the present design of Theorema, and this is now exactly where our new reasoning framework enters the stage: putting Theorema on solid foundations and thus eliminating maybe not *all*, but at least *most* of said doubt. In order to achieve this goal, our strategy is to take the well-known and established Isabelle proof assistant as some sort of 'role model' and combine the best features of Theorema and Isabelle: the strengths of Theorema have already been described above, and the key ideas behind Isabelle incorporated in our new framework are the following:

- Requiring all formulas and other expressions appearing anywhere in a formalization to be *well-formed* and *well-typed* λ-terms, for ruling out paradoxes such as Russell's. [$\rightsquigarrow$ Section 3]
- Enforcing *conservative theory extensions* by introducing new constants only via explicit non-recursive definitions, for preserving the consistency of formal theories. [$\rightsquigarrow$ Section 4]
- Restricting the inference machinery to a small kernel of *basic inferences* all proofs must go through; if these inferences are trusted to properly describe the semantics of the *constructive minimal meta logic*, then all reasoning steps in every proof conducted in this setting are automatically correct. [$\rightsquigarrow$ Section 5]
- Developing a variety of concrete object logics (like HOL, FOL, ZF, CTT) on top of the aforementioned minimal meta logic. [$\rightsquigarrow$ Section 6]

In the rest of this paper we are going to explain how we realized—or plan to realize—these ideas in the new reasoning framework.

3 Types and Terms

In this section we present the *objects* we are concerned with. The meta logic being higher-order logic, these objects are typed higher-order terms, on the one hand, and types themselves on the other hand.

3.1 Types

Fix countably infinite sets $\mathcal{A}$ and $\overline{\mathcal{B}}$, and an arbitrary set $\mathcal{B}$, such that these three sets are pairwise disjoint; assume that $\mathcal{B}$ contains at least the symbol $\rightarrow$.

Types are formed according to the grammar

$$\tau ::= \alpha \mid \overline{\gamma} \mid \gamma \mid \tau[\tau_1, \ldots, \tau_n]$$

where α is a *type variable* taken from $\mathcal{A}$, $\overline{\gamma}$ is an *arbitrary, but fixed* (a. b. f.) type constant taken from $\overline{\mathcal{B}}$,[5] and γ is type constant taken from $\mathcal{B}$.

Note that the *head* of a type, i. e. τ in the above grammar, can be completely arbitrary, and also that the number of type arguments n is independent of the head. This freedom is justified by the fact that for a proper handling of types in theorem proving it suffices to be able to check whether two type-expressions denote the same type, and to be able to unify two types by instantiating type variables. Both tasks can easily be accomplished by testing types for syntactic identity and by employing first-order syntactic unification, respectively.[6] When introducing new types, however, a more restrictive way of forming types must be adopted for preserving the consistency of the resulting theory; for instance, a new type constructor must clearly be an individual fresh constant.[7]

In the remainder of the paper we will write $\tau \to \sigma$ instead of $\to [\tau, \sigma]$, as usual, and assume that $\to$ associates to the right.

3.2 Terms

Fix countably infinite sets $\mathcal{X}$, $\mathcal{X}^*$ and $\overline{\mathcal{C}}$, as well as some arbitrary set $\mathcal{C}$, such that these four sets are pairwise disjoint.

Terms are formed according to the grammar

$$t, s ::= x \mid v^* \mid \overline{c} \mid c \mid t[s] \mid \lambda x.\ t \mid t{::}\tau$$

where x is a variable taken from $\mathcal{X}$, v^* is a *meta variable* taken from $\mathcal{X}^*$, $\overline{c}$ is an a. b. f. term constant taken from $\overline{\mathcal{C}}$, and c is a term constant taken from $\mathcal{C}$.

As can be seen, application is restricted to one argument only, which means that functions must be curried to 'simulate' higher arity.[8] As for abstractions, our implementation uses the well-known concept of de Bruijn indices to represent bound variables; their symbolic names are internally stored as well, but only for pretty-printing. $t{::}\tau$, finally, is a type annotation which entails that term t is intended to have type τ.

In order for a term to be accepted by the new reasoning framework it not only needs to be well-formed, but also *well-typed* according to the typing discipline of simply-typed λ-calculus with top-level (or *ML-style*) parametric polymorphism [10,4]. To that end, the framework contains a *Mathematica*-function that decides whether a given Theorema-term is well-typed or not based on the term's syntactical structure and on some contextual information, including declarations of constants in the background theory as well as declarations of a. b. f. constants

[5] A. b. f. type- and term constants are needed for proving, see Section 5.

[6] Unifying $\tau[\tau_1, \ldots, \tau_n]$ and $\tau'[\tau'_1, \ldots, \tau'_m]$ amounts to checking $n = m$ and unifying τ and τ', τ_1 and τ'_1, and so on.

[7] Otherwise one and the same type constructor could be defined in multiple contradictory ways.

[8] We shall still use the uncurried notation $f[a_1, \ldots, a_n]$ later on, but it must be understood as an abbreviation for $f[a_1] \ldots [a_n]$. We shall also use infix- or binder notation for certain constants whose meaning will be clear from the context.

and meta variables in proofs (c. f. Definitions 1 and 2). Here it must be noted that constants in $\mathcal{C}$ are always fully polymorphic, even when they are declared to be of a certain type in the theory: such type declarations merely serve as hints for type inference, allowing the user to omit as many explicit type annotations as possible. It could be, though, that a constant c is declared to be of type τ, but in a concrete term it occurs with type annotation τ'; then, the explicit type annotation overrules the declaration, meaning that the type of the annotated occurrence is indeed τ' rather than τ.[9] Furthermore, iterated (explicit) type annotations, as in $(t::\tau_1)::\tau_2$, merely express that the concrete type instance of t has type τ_1 *and* type τ_2 at the same time, meaning that the term only type-checks if τ_1 and τ_2 are unifiable.

All the preceding explanations exclusively refer to the *internal* representation of types and terms. It should be clear, though, that there is a lot of syntactic sugar for displaying terms in an appealing form close to usual textbook notation. Details can be found in [16].

4 Ensuring Consistency and Integrity of Formal Theories

Theorem provers in the tradition of the *Logic for Computable Functions* (LCF) systems [8] reduce the problem of proving formulas in some object logic to the problem of showing that a certain term has a certain type in the system's meta logic. In Isabelle, for instance, a formula can be proved iff it is possible to construct a meta-level object of type `thm`, the type of theorems, out of it; in such a case, the way how that object is constructed corresponds to a low-level proof of the original formula, since the type constructors of `thm` closely resemble the usual inference rules of intuitionistic higher-order logic.

We decided to pursue a different approach for our framework which, in part, is motivated by the fact that Theorema's meta language, the *Mathematica* programming language, is completely untyped; this means that anyway we cannot employ static type checking *on the meta-level* (i. e. in *Mathematica*) to mimic Isabelle's `thm`-paradigm. Instead we adopt a more ad-hoc principle of theorem proving, which allows users of the system to state any formulas they want[10] as 'facts' on the theory level, possibly using them as assumptions in the proofs of other formulas, but without having to prove these 'facts' in the first place (as long as they are unproved they are just considered axioms). But surely they *may* prove them, and in Section 5 we describe how this is accomplished.

Still, despite the quite liberal view toward formalizations expressed in the previous paragraph we *are* of course worried about the logical consistency and integrity of formal theories in Theorema. We clearly do not want to accept formalizations where half of the main theorems are stated without proof, where constant definitions possibly introduce inconsistencies, or, worse even, where the structure of the formalized theory is circular in the sense that the correctness

[9] A warning message informs the user about this perhaps unexpected result, though.

[10] As long as they are well-typed *on the object-level*, i. e. according to our own type-checker for Theorema-terms mentioned in Section 3.2

of the proof of a theorem depends on the validity of that theorem itself. To achieve this, we propose the ConsistencyChecker tool as a remedy. This tool automatically scans formalizations, recording all unproved axioms and checking whether every proved theorem indeed has a *valid* proof, whether new constants are only introduced by means of explicit, non-recursive definitions, and whether the dependency graph of all formulas in the theory is acyclic. In the end, the ConsistencyChecker reports whether all those tests succeeded, and in addition returns the set of all axioms it found. Ultimately, it is still the human users who then have to decide whether to accept the axioms and trust their being consistent, or not. [11]

So far, only a preliminary version of the ConsistencyChecker exists. It can only report unproved formulas as axioms and test dependency graphs for acyclicity, but it cannot check whether existing proofs (originally done interactively) are valid, nor whether definitions of constants form conservative extensions. Moreover, it was developed for the default set-up of Theorema, where formalizations are completely free of types; this, of course, drastically complicates the task for the ConsistencyChecker. We plan to adapt the existing version of the tool to fit with the new reasoning framework described in this paper, but this is left for future work. The interested reader may find further information on the preliminary version of the ConsistencyChecker in Section 6.4 of [9].

5 The Inference Kernel

The focus of this section exclusively lies on proving individual theorems from arbitrary sets of assumptions, without worrying about the consistency of these sets at all. Hence, here the word *proving* must always be understood as an *isolated* task within the formalization of a theory.

The inference kernel of the new reasoning framework, called Core in the actual implementation and also referred to by this name here, consists of three main components:

- A function checkProof for checking proofs, which are given as finite sequences of basic inferences and other proof methods, falling back upon
- a collection of basic inference rules and
- a function for checking well-formedness and well-typedness of terms according to Section 3,

Please note that the sole task of Core is to *check* proofs, not to find them. How a proof has originally been obtained, be it interactively by the human user or automatically by specific proof methods, is completely irrelevant for the kernel. This allows our framework to easily accommodate both interactive and automated reasoning.

In general, proving in Theorema is inherently backward-oriented: the initial proof situation, consisting of the formula that shall be proved, is gradually

[11] It should be clear that the implementation of the ConsistencyChecker, in *Mathematica*, must be trusted as well.

reduced to lists of ideally simpler proof situations, until no pending proof situations remain (success) or no further reductions are possible (failure). Reductions are realized by inference rules, which are functions that take a single proof situation, as well as some additional parameters, as input, and return a (possibly empty) list of new proof situations. Thus, Theorema adopts a quite *operational* way of looking at inference rules, rather than a *declarative* one as more common in mathematical logic and also other proof assistants, including Isabelle: there, inference rules describe how new facts can be derived from known ones, but they do not 'operate' on anything themselves: *resolution* is the driving engine behind proof in Isabelle.

The crucial difference between the default prover set-up of Theorema and Core lies in the treatment of these inference rules: in the former, inference rules can be completely arbitrary functions of said signature, implemented by the developers of the system or by end-users for their personal use; in the latter, the inference rules are fixed to a collection of 24 concrete elements[12] that are specified once and cannot be modified by users of Theorema afterward. The only thing users can (and are encouraged to) do is developing their own *proof methods*. Proof methods are similar to inference rules, in the sense that they also take a single proof situation as input, but they differ from inference rules in that they must not directly return a list of new proof situations, but only a sequence of basic inferences and/or (other) proof methods that shall be applied next. Therefore, applying proof methods in a proof in fact amounts to successively unfolding them down to sequences of basic inferences.[13] Only note that *how* proof methods are unfolded typically depends heavily on the current proof situation, so they should not be confused with mere abbreviations of fixed sequences of basic inferences. Also note that proof methods are the rough analogues of *tactics* in Isabelle, but they in fact also subsume *tacticals*: a proof method that is parametrized over other proof methods, which it applies in some particular way to the given proof situation, may be regarded a tactical.

Actually, what has been said so far is not the whole truth about proof methods in our reasoning framework. For one thing, they may not only return *one* sequence of basic inferences or other proof methods, but *arbitrarily* many of them that are interpreted as *alternatives* by the main proof-checking function. In more concrete terms, checkProof always chooses the first alternative until the whole proof is either completed or fails (due to a rule/method being not applicable), in which case it backtracks to the most recent choice it made and tries the next alternative. This concept of realizing proof search through backtracking is not new, of course; it is implemented in standard Theorema 2.0. checkProof, however, can even handle *infinitely* many alternatives that are represented as *lazy lists* (also called *streams*). This is particularly useful in connection with higher-order resolution, since higher-order unification might result in infinitely many unifiers any of which could possibly lead to successful proof.

[12] The most important of these 24 rules are listed in Section 5.2.

[13] Unfolding is not guaranteed to terminate, since proof methods might well unfold to themselves. In such a case the proof simply cannot be checked.

In addition, proof methods may even return new proof situations directly, in spite of what is written above. This is because checkProof in fact distinguishes between three *proof modes*: fast, trace and kernel. Only kernel mode imposes the strict requirements on proof methods that are described above; the other two modes do not require them at all. Apart from that, fast mode is like kernel mode, whereas trace mode, as its name suggests, not only *checks* proofs, but also *records* each individual proof step in so-called *proof objects*. Proof objects, which are a standard component of Theorema, are used to fully automatically generate natural-language proof documents, as mentioned at the beginning of Section 2. Hence, the new reasoning framework still allows the generation of such proof documents, thus preserving one of the distinctive features of Theorema.

It is the user who decides upon the proof mode a proof shall be checked w. r. t.: fast mode is typically faster than kernel mode (whence the name), making it the proof mode of choice for developing new proofs interactively in an experimental fashion. kernel mode, on the other hand, is more appropriate for checking whole formalizations with the ConsistencyChecker, to rule out all doubt about the correctness of the proofs.

Finally, before delving into more technical aspects of the inference kernel, we want to point out another feature of checkProof: not only may proofs involve global facts stated in some background theory, but also a special kind of *registered theorems*. On the surface, such theorems do not really differ from other facts, since they may simply also be used as assumptions in proofs. What really distinguishes them from other facts is that they may be *parametrized*, therefore yielding *theorem schemas*.

For preserving the correctness of proofs in the presence of such theorem schemas, checkProof demands every concrete instance of a theorem schema to come together with its proof.[14] All instances that are made use of in a proof, or in a series of proofs, are recorded, and the proofs of these instances (of which there can only be finitely many) may then be checked, too.

Example 1. In a nested implication, like $A_1 \longrightarrow A_2 \longrightarrow \ldots \longrightarrow A_n \longrightarrow C$, the order of the antecedents $A_1, \ldots, A_n$ is irrelevant, since all possible arrangements yield equivalent formulas (if $\longrightarrow$ is interpreted in the usual way). In higher-order predicate logic, however, this fact cannot be expressed by a single theorem, but only by a theorem schema of the form

$$(A_{i_1} \longrightarrow \ldots \longrightarrow A_{i_n} \longrightarrow C) \longrightarrow (A_1 \longrightarrow \ldots \longrightarrow A_n \longrightarrow C)$$

where i is an arbitrary permutation of $\{1, \ldots, n\}$ and where the A_j and C are tacitly assumed to be universally quantified. Our new reasoning framework provides a *Mathematica*-function that, given a permutation i, returns precisely the corresponding instance of the theorem schema together with the formal proof of precisely that instance.

[14] In other words: theorem schemas must be proved by proof schemas.

5.1 Proof Situations

So far we have sketched the overall proof-checking and, to some extent, also searching procedure in Core. In this subsection we now want to shed some light on the objects a Core proof actually manipulates: *proof situations*. The term 'proof situation' is non-standard in common literature on mathematical logic or mechanized reasoning, but it is used in Theorema to refer to objects that would elsewhere be called *subgoals* instead. We also use the term 'subgoals' in the remainder of this section, but with a slightly different meaning that will become clear soon. Also note that, although the proof situations we consider here are roughly the same as the proof situations in the default prover set-up of Theorema, there are some differences, too; we do not list them here explicitly, though.

Before we can define proof situations formally, we need some auxiliary notions.

Definition 1. *A* type context Γ *for a. b. f. constants is a (finite) set of type annotations* $\overline{c}::\tau$*, where* $\overline{c} \in \overline{\mathcal{C}}$ *is an a. b. f. term constant,* τ *is a type, and all such a. b. f. term constants in* Γ *are distinct.*

We shall say that $\overline{c}$ *is contained in* Γ *if* $\overline{c}::\tau \in \Gamma$*, for some* τ*.*

Definition 2. *A* type- and dependency context Ξ *for meta variables is a (finite) set of pairs* $(v^*::\tau, \Gamma)$*, where* $v^* \in \mathcal{X}^*$ *is a meta variable,* τ *is a type,* Γ *is some typing context for a. b. f. constants ("an instance of* v^* *may only involve the a. b. f. constants in* Γ*"), and all such meta variables in* Ξ *are distinct.*

We shall say that v^* *is contained in* Ξ *if* $(v^*::\tau, \Gamma) \in \Xi$*, for some* τ *and* Γ*.*

Intuitively, the meaning of a. b. f. constants and meta variables should be clear: the former are introduced in proofs when the proof goal is universally quantified and, hence, shall be proved for some 'arbitrary, but fixed' values, and the latter typically stem from universally quantified assumptions that shall be instantiated by some term yet to be synthesized in the course of the proof; the dependency information then specifies which of the a. b. f. constants the term may depend upon.[15] This will ensure that instances of meta variables may only involve a. b. f. constants introduced *before* the respective meta variables, thanks to inference rule meta presented in Section 5.2.

Definition 3 (Proof situation). *A* proof situation *is characterized by a finite sequence of subgoals* G*, a finite sequence of hypotheses* H*, a type context* Γ *for a. b. f. term constants, and a type- and dependency context* Ξ *for meta variables. A proof situation is well-formed iff*

1. *the elements of* G *and* H *are well-typed closed terms of type FORM, the type of* Core *formulas,*[16] *where* Γ *and* Ξ *specify the types of a. b. f. term constants and meta variables, respectively,*

[15] Meta variables are analogous to schematic variables in Isabelle, but the dependence of schematic variables on a. b. f. constants is made explicit in Isabelle by applying them to the a. b. f. constants they may depend on.

[16] Corresponds to *prop* in Isabelle.

2. *H is free of meta variables, and*
3. *the annotated type (in Γ) of every a. b. f. constant occurring in H is free of type variables.*

Well-formed proof situations are denoted by $H \vdash_{\Gamma,\Xi} G$.

Just as for a. b. f. constants and meta variables, the meaning of a proof situation $H \vdash_{\Gamma,\Xi} G$ should intuitively be clear: it describes a situation which, in order to establish the validity of the formula we want to prove originally, must be *closed* by showing that the hypotheses in H imply *all* formulas in G.[17] The types in G, H, Γ and Ξ may depend on a. b. f. type constants $\overline{\gamma}_1, \ldots, \overline{\gamma}_i$ and type variables $\alpha_1^* \ldots, \alpha_j^*$.

The following is not meant to be a precise formal description of the semantics of proof situations, but it hopefully conveys the idea about the roles of the various constituents of proof situations: a well-formed proof situation

$$h_1, \ldots, h_k \vdash_{\{\overline{c}_1::\sigma_1,\ldots,\overline{c}_m::\sigma_m\},\{(v_1^*::\tau_1,\Gamma_1),\ldots,(v_n^*::\tau_n,\Gamma_n)\}} g_1, \ldots, g_\ell$$

may be associated with the *intuitionistic* higher-order predicate logic formula

$$\begin{aligned}\forall_{\overline{\gamma}} \exists_{\alpha^*,v^*} \forall_{\overline{z},\overline{c}} \quad & (\forall_{\alpha^*} h_1(\overline{\gamma},\alpha^*,\overline{z}) \wedge \ldots \wedge \forall_{\alpha^*} h_k(\overline{\gamma},\alpha^*,\overline{z})) \Rightarrow \\ & (g_1(\overline{\gamma},\alpha^*,\overline{z},\overline{c},v^*(\Gamma)) \wedge \ldots \wedge g_\ell(\overline{\gamma},\alpha^*,\overline{z},\overline{c},v^*(\Gamma))) \qquad (1)\end{aligned}$$

where the dependence of the h_i and g_i on the various type- and term constants and -variables, according to the well-formedness constraints on proof situations, is made explicit, and where

- $\overline{\gamma}$ and α^* abbreviate the sequence of all a. b. f. type constants and type variables, respectively (note that type variables are not shared among the hypotheses and the subgoals, because they are quantified universally in each hypothesis),
- $\overline{z}$ and $\overline{c}$ partition the a. b. f. constants $\overline{c}_1, \ldots, \overline{c}_m$ into two disjoint sequences such that those contained in $\overline{z}$ are precisely those whose types do not depend on any type variables,
- v^* abbreviates $v_1^*, \ldots, v_m^*$, and
- $v^*(\Gamma)$ abbreviates $v_1^*(\Gamma_1), \ldots, v_n^*(\Gamma_n)$, expressing that the instance of v_1^* may depend on the a. b. f. constants in Γ_1, etc.

At any point in a proof there may be several pending proof situations, since proving proceeds by reducing one proof situation to a *list* of other proof situations, as explained above; hence, *all* of them have to be closed in order to complete the proof. Different proof situations are always completely independent of each other, in the sense that instantiating term meta variables (or type variables) in one of them does not affect term meta variables (or type variables) in others, even if the variables by chance have the same name. Subgoals within

[17] In contrast to sequent calculus, where it suffices to show that *one* of the formulas on the right-hand-side of the turnstile is a consequence of the hypotheses.

one single proof situation, on the other hand, *do* share such variables (and actually also the hypotheses), hence they are not independent of each other. This, in fact, is the main reason for distinguishing proof situations and subgoals.

The proof of some formula g starts with one initial proof situation, which is of the form $\vdash_{\emptyset,\emptyset} \ \overline{g}$, where $\overline{g}$ denotes the *type specialization* of g, i. e. g with all type variables replaced by fresh a. b. f. type constants. This specialization ensures that *all* type instances of the original formula are proved.

5.2 Inference Rules

The inference rules presented below define the semantics of the constructive higher-order meta logic of the inference kernel. Just like the meta logic of Isabelle, its language consists of abstraction (λ), implication ($\longrightarrow$), universal quantification ($\bigwedge$) and equality ($\approx$) on the term level, and of a type of formulas ($FORM$) and function types ($\rightarrow$) on the type level. The inference rules define the semantics of the logic by specifying how proof situations built from terms in the aforementioned language can be manipulated, which in our case means how they can be reduced to zero, one or more new proof situations.

What follows is the list of basic inference rules of Core. Note that the rules have to be read bottom-up: when constructing a proof, proof situations *below* the horizontal line are reduced to proof situations *above* the horizontal line. This reading also motivates the names of makeExplicit and makeImplicit, which would be misleading otherwise. In a sequence like h, H in the hypotheses of a proof situation, h stands for some hypothesis and H for the (possibly empty) sequence of remaining hypotheses; subgoals are handled analogously.

- close:

$$\frac{}{H \ \vdash_{\Gamma,\Xi} \ \emptyset}$$

- makeExplicit:

$$\frac{h, H \ \vdash_{\Gamma,\Xi} \ h' \longrightarrow g, G}{h, H \ \vdash_{\Gamma,\Xi} \ g, G}$$

  where h' is the 'lifted' version of h, meaning that all type variables in h' were renamed to avoid clashes with type variables in $\{g, G\}$.
- makeImplicit:

$$\frac{a, H \ \vdash_{\Gamma,\Xi} \ a \longrightarrow b}{H \ \vdash_{\Gamma,\Xi} \ a \longrightarrow b}$$

  on the proviso that a does not depend on any meta variables or type variables.[18]
- drop:

$$\frac{H \ \vdash_{\Gamma,\Xi} \ b, G}{H \ \vdash_{\Gamma,\Xi} \ a \longrightarrow b, G}$$

[18] Note that a still remains among the explicit assumptions of the first and only subgoal even after adding it to the hypotheses.

– split:

$$\frac{H \ \vdash_{\Gamma,\Xi} \ g \qquad H \ \vdash_{\Gamma,\Xi} \ G}{H \ \vdash_{\Gamma,\Xi} \ g, G}$$

on the proviso that g does not share any term meta variables and type variables with G.

– cut:

$$\frac{H \ \vdash_{\Gamma,\emptyset} \ \widetilde{a} \qquad a, H \ \vdash_{\Gamma,\Xi} \ G}{H \ \vdash_{\Gamma,\Xi} \ G}$$

on the proviso that a type-checks to $FORM$, that it does not contain any meta variables, and that all a. b. f. term constants appearing in a are independent of type variables. $\widetilde{a}$ is the type specialization of a, i. e. a with all type variables replaced by fresh a. b. f. type constants.

– meta:

$$\frac{H \ \vdash_{\Gamma,\Xi \cup \{(v^* :: \tau, \Gamma)\}} \ G}{H \ \vdash_{\Gamma,\Xi} \ G}$$

on the proviso that v^* is a meta variable not occurring in Ξ and τ is a type.

– inst:

$$\frac{H \ \vdash_{\Gamma,\Xi} \ G(t)}{H \ \vdash_{\Gamma,\Xi \cup \{(v^* :: \tau, \Gamma_0)\}} \ G(v^*)}$$

on the proviso that t type-checks to τ and that all a. b. f. term constants in t are contained in Γ_0.

– instT:

$$\frac{H \ \vdash_{\Gamma(\tau),\Xi(\tau)} \ G(\tau)}{H \ \vdash_{\Gamma(\alpha^*),\Xi(\alpha^*)} \ G(\alpha^*)}$$

where α^* is an arbitrary type variable and τ is a (potentially polymorphic) type. $G(\alpha^*)$, $\Gamma(\alpha^*)$ and $\Xi(\alpha^*)$ express that α^* may occur in G, Γ and Ξ, and that is has to be instantiated by τ everywhere.

– allE:

$$\frac{H \ \vdash_{\Gamma,\Xi} \ a(t) \longrightarrow b, G}{H \ \vdash_{\Gamma,\Xi} \ (\bigwedge x :: \tau.\ a(x)) \longrightarrow b, G}$$

on the proviso that t type-checks to τ.

– allI:

$$\frac{H \ \vdash_{\Gamma \cup \{\overline{c} :: \tau\},\Xi} \ g(\overline{c}), G}{H \ \vdash_{\Gamma,\Xi} \ \bigwedge x :: \tau.\ g(x), G}$$

on the proviso that $\overline{c}$ is a fresh a. b. f. term constant that does not appear in Γ.

– impE:[19]

$$\frac{H \ \vdash_{\Gamma,\Xi} \ [a_1 \longrightarrow \ldots \longrightarrow a_m \longrightarrow p_i \,|\, 1 \leq i \leq n], G}{H \ \vdash_{\Gamma,\Xi} \ (p_1 \longrightarrow \ldots \longrightarrow p_n \longrightarrow g) \longrightarrow a_1 \longrightarrow \ldots \longrightarrow a_m \longrightarrow g, G}$$

[19] impE *backchains* a subgoal w. r. t. an implication among the assumptions of that subgoal: each antecedent p_i of the implication gives rise to a new subgoal.

– refl:

$$\frac{H \ \vdash_{\Gamma,\Xi} \ G}{H \ \vdash_{\Gamma,\Xi} \ a \approx b, G}$$

if a is α-equivalent to b.[20]

– replace:

$$\frac{H \ \vdash_{\Gamma,\Xi} \ a \approx b \longrightarrow g(b), G}{H \ \vdash_{\Gamma,\Xi} \ a \approx b \longrightarrow g(a), G}$$

– alphabetaeta:

$$\frac{H \ \vdash_{\Gamma,\Xi} \ G_{p \to t}}{H \ \vdash_{\Gamma,\Xi} \ G}$$

where p is the position of an arbitrary subexpression e of G (possibly containing free variables), t is a term $\alpha\beta\eta$-equivalent to e, and $G_{p \to t}$ denotes G with e replaced by t.[21]

– ext:

$$\frac{H \ \vdash_{\Gamma,\Xi} \ \bigwedge x_1{::}\tau_1 \ldots x_n{::}\tau_n.\ a[x_1, \ldots, x_n] \approx b[x_1, \ldots, x_n], G}{H \ \vdash_{\Gamma,\Xi} \ a \approx (b{::}\tau_1 \to \ldots \to \tau_n \to \sigma), G}$$

Apart from the rules listed above, there are some other rules that merely serve organizational needs. For instance, there are rules for inserting facts from the background theory into the hypotheses of a proof situation, and for rearranging the list of subgoals of a proof situation by moving an arbitrary subgoal to the front. We do not go into further detail here, since these rules are only of minor relevance.

The set of inference rules is not the smallest theoretically possible, and even the rules themselves could in some cases be weakened. For instance, it would suffice if allE instantiated universal quantifiers by fresh meta variables only; inst could then be used to instantiate these meta variables by concrete terms. Or, in ext it would be sufficient if only one function type was considered at a time. The goal we want to achieve with the particular choice and formulation of the inference rules above is a reasonable trade-off between *simplicity* on the one hand, and *efficiency* (in terms of the time needed to check proofs) on the other hand. Indeed, first experiments suggest that performance is really going to be an issue in larger proofs.

Finally, we want to draw the reader's attention to the following important observation: none of the inference rules require higher-order unification. Therefore, although our new reasoning framework features a general higher-order pre-unification procedure for automatically finding suitable instances of meta variables, it is *not* part of the kernel. This further implies that said procedure—implemented correctly or not—does not influence the correct behavior of the

[20] Using de Bruijn indices in the concrete implementation, α-equivalence tests amount to checking syntactic identity.

[21] alphabetaeta cannot be replaced by a combination of replace and refl, since the formula $a \approx b$ in the latter two rules must not contain free variables (as every formula in a proof situation). alphabetaeta, on the other hand, may convert terms even if they contain variables bound somewhere outside.

inference kernel at all, and it further implies that we could safely connect it to external, highly optimized unification tools in the future.

6 Object Logics

In the preceding section we described the meta logic of Core, but we have not said anything about any *object logic* yet. Of course, one could simply formalize theories in that meta logic, maybe making it classical by introducing the law of the excluded middle (or anything equivalent to it) as an additional axiom. A probably more reasonable alternative, however, is to *present* object logics within the meta logic through *judgments* that 'translate' statements from the object logic into the meta logic; this is precisely the approach pursued by Isabelle, for instance [12]. The semantics of an object logic is then defined entirely by a set of axioms expressing the characteristic properties of the respective judgments.

The new reasoning framework allows to present object logics in the way sketched above, thus retaining a good deal of the flexibility of Theorema for the user who can formalize his mathematics in the logic of his choice. Even more, the very design of the framework already anticipates the presence of all kinds of object logics, with potentially completely different syntax and notation, by setting up a *programmable* interface between the GUI of Theorema and the inference kernel Core. 'Programmable' means here simply that after parsing expressions w. r. t. the default parsing rules of Theorema and *Mathematica* the results are passed to a *translator* function that may apply further transformations—and this translator function can be implemented by an object logic.

But not only parsing and pretty-printing, but also reasoning can greatly be influenced by object logics, simply by adding specialized proof methods of all sorts to the arsenal of proof methods provided by Core by default. For instance, one of the most important proof methods for logics with equality is a *simplifier* that rewrites subgoals w. r. t. known (quantified, conditional) equalities. But there might also be proof methods that merely apply suitable instances of certain basic facts as backward rules (c. f. inference impE) to the current proof situation, saving the user from picking and instantiating the facts herself.

So far our work mainly concentrated on implementing Core and developing general-purpose Core proof methods that are independent of any object logic. Setting up classical higher-order logic as the first object logic of our new framework, analogous to Isabelle/HOL, is currently work in progress.

7 Conclusion

We presented a new reasoning framework for Theorema 2.0 whose main objective is to increase the trust human users can have in the theories formalized in Theorema. Even though the fundamental components of the framework are already there, lots of work remains in developing sophisticated proof methods

that automate proof search as much as possible, and in formalizing enough elementary concepts of at least one object logic (e. g. HOL) such that one can reasonably start building 'more interesting' formalizations upon them.

Acknowledgments. Thanks are due to Temur Kutsia and David Cerna for their feedback on a draft version of this paper. I also thank the anonymous referees for their valuable comments.
The research was funded by the Austrian Science Fund (FWF): P 29498-N31.

References

1. Buchberger, B.: Towards the Automated Synthesis of a Gröbner Bases Algorithm. Revista de la Real Academia de Ciencias, Serie A: Mathematicas 98(1), 65–75 (2004)
2. Buchberger, B., Crăciun, A., Jebelean, T., Kovács, L., Kutsia, T., Nakagawa, K., Piroi, F., Popov, N., Robu, J., Rosenkranz, M., Windsteiger, W.: Theorema: Towards Computer-Aided Mathematical Theory Exploration. J. Applied Logic 4(4), 470–504 (2006)
3. Buchberger, B., Jebelean, T., Kutsia, T., Maletzky, A., Windsteiger, W.: Theorema 2.0: Computer-Assisted Natural-Style Mathematics. J. Formalized Reasoning 9(1), 149–185 (2016)
4. Cardelli, L.: Basic Polymorphic Typechecking. Science of Computer Programming 8(2), 147–172 (1987)
5. Craciun, A.: Lazy Thinking Algorithm Synthesis in Gröbner Bases Theory. Ph.D. thesis, RISC, Johannes Kepler University Linz (2008)
6. Geuvers, H.: Proof Assistants: History, Ideas and Future. Sadhana Journal 34(1), 3–25 (2009)
7. Giese, M., Buchberger, B.: Towards Practical Reflection for Formal Mathematics. RISC Report Series 07-05, RISC, Johannes Kepler University Linz (2007)
8. Gordon, M.J., Milner, A.J., Wadsworth, C.P.: Edinburgh LCF: A Mechanised Logic of Computation, LNCS, vol. 78. Springer (1979)
9. Maletzky, A.: Computer-Assisted Exploration of Gröbner Bases Theory in Theorema. Ph.D. thesis, RISC, Johannes Kepler University Linz (May 2016)
10. Milner, R.: A Theory of Type Polymorphism in Programming. J. Computer and System Sciences 17(3), 348–375 (1978)
11. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
12. Paulson, L.C.: Isabelle: The next 700 Theorem Provers. In: Odifreddi, P. (ed.) Logic and Computer Science. pp. 361–386. Academic Press (1990)
13. Rosenkranz, M.: New Symbolic Method for Solving Linear Two-Point Boundary Value Problems on the Level of Operators. J. Symbolic Computation 39(2), 171–199 (2004)
14. Wenzel, M.: Isabelle/Isar Reference Manual (2017), part of the Isabelle documentation
15. Wiedijk, F. (ed.): The Seventeen Provers of the World, LNAI, vol. 3600. Springer (2006)
16. Windsteiger, W.: Theorema 2.0: A Graphical User Interface for a Mathematical Assistant System. In: Kaliszyk, C., Lueth, C. (eds.) UITP'2012. EPTCS, vol. 118, pp. 72–82. Open Publishing Association (2012)