Theorema-HOL: Classical Higher-Order Logic in Theorema

Alexander Maletzky*

RISC, Johannes Kepler Universität Linz, Austria,
alexander.maletzky@risc.jku.at

Abstract

We present Theorema-HOL, an add-on package for the Theorema 2.0 proof assistant which enables users to formalize their mathematical theories in classical higher-order logic. Besides the logical axioms, definitions, theorems, etc. needed for that purpose, Theorema-HOL also comes with an intuitive and easy-to-use theory- and proof language, inspired by Isabelle/Isar, for proving theorems interactively. Furthermore, Theorema-HOL, and its underlying framework Theorema-Core, force formalizations to conform to certain well-defined standards as a means to establishing their logical correctness and consistency.

1 Introduction

Theorema 2.0 [4] is a mathematical assistant system based on *Mathematica* [23], meant to support mathematicians, logicians and computer scientists in many areas of their everyday work. It is similar to other projects of that kind, like Isabelle [16], Coq [3], Mizar [2], and others, but it differs from these systems in two important aspects: (i) Theorema focuses on *natural style* mathematics that should be accessible to mathematicians who are no experts in the use of proof assistants in general and Theorema in particular, and (ii) Theorema is a very *flexible* system that imposes as few constraints, regarding the way how mathematics is to be presented, on its users as possible. While both these features are certainly desirable, the second bears some potential dangers, too, as one of the main objectives of proof assistants is to enforce the formalization of mathematics in a *well-defined*, *consistent* manner, such that users can fully rely on the correctness and consistency of existing formal theories when building upon them.

To that end, we developed Theorema-Core [13] as an add-on package for Theorema which addresses precisely the aforementioned shortcoming: Theorema-Core implements a small logical kernel of constructive, simply-typed higher-order logic that ensures the integrity (wrt. this well-defined logic) of theories formalized in it. "Integrity" in this context does not mean that each formalized theory indeed has a model – after all users may still state arbitrary, possibly contradictory axioms – but at least the system tries to detect and rule out as

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

many sources of inconsistency as possible; this, above all, concerns extensions of theories by explicit definitions of new types and constants. More information on Theorema-Core can be found in Section 3.1.

Due to the constructive nature of Theorema-Core and the fact that most of modern-day mathematics is formulated in a classical setting, we recently also developed Theorema-HOL as the classical analogue of Theorema-Core; more precisely, Theorema-HOL is classical higher-order logic formulated in the logic of Theorema-Core. In short, Theorema-Core is the meta-logic of Theorema-HOL, and the precise relationship between Theorema-HOL and Theorema-Core is discussed in Section 3.2. Readers familiar with Isabelle will realize that the situation here parallels that of Isabelle/Pure [18] vs. Isabelle/HOL [16] in many respects. In fact, various ideas incorporated into Theorema-Core and Theorema-HOL were originally borrowed from Isabelle.

The report is organized as follows: Section 2 gives an impression of how mathematical theories can be formalized in the framework of Theorema-HOL, without explaining any underlying technicalities. Section 3 details the theoretical background of both Theorema-Core and Theorema-HOL, and their precise relationship. Section 4 compares Theorema-HOL to standard-Theorema, and Section 5 compares it to Isabelle/HOL; these two sections assume some familiarity with the respective systems and may hence be skipped by readers only interested in Theorema-HOL itself. Section 6, finally, concludes the report.

2 Formalizing Mathematics in Theorema-HOL

This section aims at giving an informative overview about how Theorema-HOL can effectively be used to formalize mathematical theories. The sample formalization presented below, *elementary set theory*, is available online as part of the 'Core+HOL+Interactive' add-on package for Theorema 2.0:[1] notebook 'CoreTheories/HOL/Sets.nb'. Note, however, that compared to the actual formalization we simplify certain technicalities in this exposition for the sake of better readability; for instance, we use object-level logical quantifiers and connectives throughout, even where in reality their meta-level analogues occur.

Remark 1. Further sample notebooks about using Theorema-HOL can be found in the 'Documentation/' subfolder of the 'Core+HOL+Interactive' package.

2.1 Introducing New Types

All starts with introducing the right types for the concepts one wishes to formalize. In the case of set theory, the objects of interest are, of course, *sets*, so we employ Theorema-HOL's *type definition* mechanism for creating a type of sets. Indeed, since a set of elements of some type α can be thought of a unary predicate of type $\alpha \rightarrow \mathsf{BOOL}$ that, for every given element decides whether it is included in the set, the type of sets is nothing else than an *isomorphic copy* of the existing type $\alpha \rightarrow \mathsf{BOOL}$.[2] In Theorema-HOL, this desired copy is constructed by issuing the command

[1] `www.risc.jku.at/projects/tetra-gb/AddOns.html`

[2] The type system underlying Theorema-HOL only supports so-called *top-level polymorphism*, meaning that all elements of a set must have the same type.

typedef '$\mathsf{SET}[\alpha\_]$' := '$\alpha\_ \to \mathsf{BOOL}$' (**morphisms** 'Collect' 'Contains')

which introduces the new unary type constructor SET and registers the *abstraction- and representation morphisms* Collect and Contains, respectively.

Before we can explain more thoroughly what the above command does, some syntactical details must be clarified:

- In Theorema-Core commands, types and terms have to be written inside single quotation marks to distinguish them from elements of the command language.
- Type variables are indicated by an underscore-suffix, to distinguish them from type constructors. We take the liberty to omit this suffix in informal text, when it is clear that a certain identifier (usually a Greek letter) denotes a type variable.
- Following usual *Mathematica* syntax, function application is denoted by square brackets. This also applies to applications of type constructors.

After issuing the above command, Theorema-HOL knows that SET is a unary type constructor that can henceforth be used in definitions of other types, constant definitions, theorem statements, etc. But in addition to that, the system also knows its exact relationship with the type $\alpha \to \mathsf{BOOL}$ via the two type morphisms Collect and Contains: Collect is a constant of type $(\alpha \to \mathsf{BOOL}) \to \mathsf{SET}[\alpha]$ that converts predicates into the sets they represent, and Contains is a constant of type $\mathsf{SET}[\alpha] \to \alpha \to \mathsf{BOOL}$ that converts sets into the predicates by which they are represented. Hence, Collect and Contains are inverses of each other, which is expressed by the two automatically generated characteristic axioms

$$\underset{A}{\forall}\ \mathsf{Collect}[\mathsf{Contains}[A]] = A \qquad (\textit{Contains inverse})$$

$$\underset{P}{\forall}\ \mathsf{Contains}[\mathsf{Collect}[P]] = P \qquad (\textit{Collect inverse})$$

based on which a couple of theorems about the morphisms (e. g. injectivity) are automatically proved, too.[3]

So much about the definition of the type of sets. Before we show how new constants, e. g. the membership relation $\in$, can be introduced, we want to emphasize that it is not only possible to introduce new types as isomorphic copies of existing types, but also as *non-empty subsets* of existing types. For instance, assume we have already defined a predicate isFinite that determines whether its argument is finite. Then the type of finite sets would be defined by the command

typedef '$\mathsf{FSET}[\alpha\_]$' $\simeq$ '$\mathsf{isFinite} :: (\mathsf{SET}[\alpha\_] \to \mathsf{BOOL})$'

where one must in particular note that compared to the command above, instead of ":=" there is "$\simeq$" and that on the right-hand-side of "$\simeq$" there is a (type-annotated) predicate rather than a type. The semantics of the command should be clear: introduce a new type constructor FSET such that $\mathsf{FSET}[\alpha]$ is isomorphic to the class of finite sets of type $\mathsf{SET}[\alpha]$. As for the morphisms

[3]They are not important for the present case of sets, so we omit them here.

converting between FSET and SET, we could again have specified their names explicitly, but since we omitted the optional **morphisms**-clause their names default to $\mathsf{FSET{\smile}abs}$ and $\mathsf{FSET{\smile}rep}$, respectively.[4]

However, in higher-order logic all types must be inhabited, which means that when defining FSET as above, one must also prove that there *exists* a finite set, i. e. a set satisfying $\mathsf{isFinite}$. So, when evaluating the command, Theorema automatically adds a new formula (unless it is already present) to the theory notebook that expresses precisely said fact; in the current case it obviously is

$$\underset{A}{\exists}\ \mathsf{isFinite}[A] \qquad (\textit{FSET not empty})$$

At the beginning, (*FSET not empty*) is a mere axiom, but of course we may at any time prove it and therefore turn it into a theorem; how theorems can be proved in Theorema-HOL is explained in Section 2.3. Note that SET does not require any such non-emptiness assertion, because $\mathsf{SET}[\alpha]$ is a full copy of an existing type, which is already known to be inhabited.

Finally, the characteristic axioms of $\mathsf{FSET{\smile}abs}$ and $\mathsf{FSET{\smile}rep}$ are

$$\underset{A}{\forall}\ \mathsf{isFinite}[\mathsf{FSET{\smile}rep}[A]] \qquad (\textit{FSET}{\smile}\textit{rep})$$

$$\underset{A}{\forall}\ \mathsf{FSET{\smile}abs}[\mathsf{FSET{\smile}rep}[A]] = A \qquad (\textit{FSET}{\smile}\textit{rep inverse})$$

$$\underset{A}{\forall}\ \mathsf{isFinite}[A] \Rightarrow \mathsf{FSET{\smile}rep}[\mathsf{FSET{\smile}abs}[A]] = A \qquad (\textit{FSET}{\smile}\textit{abs inverse})$$

2.2 Introducing New Constants

Having SET and its corresponding morphisms $\mathsf{Collect}$ and $\mathsf{Contains}$, it is straightforward to define the membership relation for sets. Namely, $\mathsf{Contains}$ is already almost exactly what we need: it takes a set and an element and determines whether the element is in the set; only the order of arguments is wrong. Therefore, our definition of $\mathsf{Element}$ (infix $\in$) is simply the formula

$$\underset{a,A}{\forall}\ \mathsf{Element}[a, A] :\Leftrightarrow \mathsf{Contains}[A, a] \qquad (\textit{Element def})$$

So, definitions of constants are not achieved through commands (as are type definitions), but by stating the definitions as ordinary Theorema formulas, which must be (universally quantified) equalities/equivalences, where, however, the main relation symbol must be $:=$ (or $:\Leftrightarrow$) instead of $=$ (or $\Leftrightarrow$) – only then the formulas are really detected to be *definitions* and not just arbitrary axioms or theorems. Of course, from the purely logical point of view, a definition *is* an axiom, but on the other hand it is also clear that a definition represents a different kind of axiom. Intuitively, definitions are mere *abbreviations* that could, in principle, be eliminated altogether. This in particular implies that a definition – if done properly – will never add an inconsistency to the logic, and therefore the system should check whether definitions *are* done properly. The precise meaning of *proper* in this context is technical, but intuitively it means that definitions should neither be *overlapping* (two definitions for the same constant) nor *circular* (definition of constant depends on that constant). As experience

[4]Underscores have a special meaning in *Mathematica*, so $\smile$ is used instead to construct compound identifiers.

in Isabelle/HOL shows [11], ensuring these two crucial properties of definitional theory extensions is by far not as easy as it seems at first glance, especially when allowing *constant overloading* (i. e. providing different definitions for different type instances of a constant, which is possible both in Isabelle/HOL and in Theorema-HOL, see below). The decision algorithm we implemented in our system is basically the same as the one in Isabelle/HOL, described in detail in [12].

The fact that $a \in A$ is the usual infix notation for $\mathsf{Element}[a, A]$ (both for parsing and pretty-printing) comes for free, because it is "hard-wired" in the underlying *Mathematica* system. Unfortunately, this also means that neither can built-in infix notations be switched off, nor can new infix notations be added easily – one basically has to live with what *Mathematica* provides by default (which is in most cases fairly reasonable). What *can* be customized easily, however, is the way how *sequences* of binary operators, like $\in$, are interpreted. For instance, in mathematics it is common to write $a \in A \in \mathcal{A}$ as a shortcut for $a \in A \wedge A \in \mathcal{A}$. In Theorema-HOL, introducing such a shortcut is as easy as can be: it is achieved by the command

notation 'Element': **binary** 'And'

which informs Theorema-HOL that sequences of $\in$ shall be combined (during parsing) using the logical connective $\wedge$. Arbitrary other combinators may be given instead of $\wedge$, as long as their types fit. Furthermore, notations can not only be specified for terms, but analogously also for types; we omit the details here.

Next, one might think of introducing the negation of set membership, $\notin$, as the obvious abbreviation for $\neg \in$. To that end, we could proceed as before and define NotElement (which is the long name of $\notin$ in *Mathematica*) as a new constant. But in this case it is more reasonable to regard $\notin$ as mere extra-logical *syntactic sugar* the reasoning system never gets to see internally, i. e. an abbreviation that is always unfolded to $\neg \in$ already during parsing. Theorema-HOL enables this by means of the **abbreviation** command:

abbreviation 'NotElement' $\rightleftharpoons$ '$\underset{a,A}{\lambda} \neg a \in A$'

adds corresponding parsing- and pretty-printing rules that transform occurrences of $\notin$ into $\neg \in$ and back again. Similar as **notation**, **abbreviation** also works for types. If $\rightleftharpoons$ is replaced by $\rightharpoonup$, the abbreviation is only unfolded during parsing, but not folded back during pretty-printing; with $\leftharpoondown$ it is just the other way round.

Remark 2. The reason why abbreviations are not introduced in formulas, like definitions, is that then the symbols $\rightleftharpoons$, $\rightharpoonup$ and $\leftharpoondown$ would already have a predefined meaning in the term language, i. e. could not be used otherwise any more. In the current setup, they only have a meaning in the command language.

Apart from definitions and abbreviations, constants can be introduced axiomatically as well: first declare them and then state their characteristic properties as axioms. In fact, precisely this happens for the type morphisms (cf. Section 2.1): Collect, for instance, is neither introduced by a definition nor an abbreviation, but solely by means of its characteristic properties (*Collect inverse*) and (*Contains inverse*). Since axiomatizations have the potential to add inconsistencies to the theory, users should be very careful when using them, or avoid them altogether. But anyway, declaring a constant without defining it is accomplished by a command like

decl '$\mathsf{SquareIntersection} :: (\alpha_{\_} \to \alpha_{\_} \to \alpha_{\_})$'

i. e. the keyword **decl** followed by one or more type-annotated constants that have not been declared yet; note here that definitions à la (*Element def*) also automatically declare the constants to be defined *unless they have been declared already.* Continuing the example of SquareIntersection (infix $\sqcap$), we could now specify its meaning axiomatically as indicated above, but we could also still *define* the constant – and even give different definitions for different type instances, i. e. *overload* it! Typically, $\sqcap$ is used for the "meet" operation in lattices, and since both BOOL and $\mathsf{SET}[\alpha]$ are lattices, we can define $\sqcap$ for these two types by stating the two definitions

$$\underset{P,Q}{\forall}\ P \sqcap Q :\Leftrightarrow P \wedge Q \qquad (\sqcap\ \mathit{BOOL\ def})$$

$$\underset{A,B}{\forall}\ A \sqcap B := \mathsf{Collect}[\underset{x}{\lambda}\, x \in A \wedge x \in B] \qquad (\sqcap\ \mathit{SET\ def})$$

Theorema-HOL accepts overloaded definitions, as long as the types of the respective type instances of the constants are not unifiable after renaming type variables apart; see again [12] for details. Note that in the two definitions of $\sqcap$ it is not necessary to explicitly type-annotate the bound variables or the occurrences of $\sqcap$, because the right types can be inferred by the system automatically.

Remark 3. It is possible to *lock* constants: a locked constant cannot be defined. This is particularly useful when specifying constants axiomatically, because then it should not be possible any more to also give definitions for them. Type morphisms are always locked.

2.3 Proving Theorems

The main task of a proof assistant is, of course, to prove theorems. Or at least to *support* the user in proving theorems by checking the correctness of user-generated proofs on the spot and automating as many aspects of the proof search as possible. Historically, Theorema has always put the emphasis on fully automatic "push-button" proving: the user only needs to specify which formula to prove and which knowledge to use in the proof, and the system then either manages to find a proof automatically or fails entirely. In Theorema-Core and Theorema-HOL, however, we decided to focus on *interactive* theorem proving: the user has to compose the proof in a particular formal language, and the system *checks* whether the given proof is indeed valid, i. e. conforms with the very logic of the reasoning system. Besides, the system still provides powerful *proof methods* that are able to prove at least *some* (sufficiently simple) subgoals automatically; see below for a concrete example.

The proof language of Theorema-Core closely resembles Isabelle/Isar [20]: the main proof commands of Isar, like **proof**, **fix**, **assume**, **have** etc., are also present in Theorema-HOL and have a similar syntax and semantics there, as can be seen below. The intention behind the close resemblance of Isar and Theorema-Core's proof language is twofold:

- Isar is generally well-thought. It does not make sense to come up with a completely different proof language only for the sake of being different.

- Many people are familiar with Isabelle, and in particular with Isar, and they can thus get acquainted with the proof language of Theorema-Core easily.

Let us now look at a concrete example: we illustrate the process of interactive proving in Theorema-HOL by means of the well-known *Cantor paradox* which states that there is no surjective function from a set to its power-set:[5]

$$\nexists_{f::(\alpha_{\_}\to\mathsf{SET}[\alpha_{\_}])} \forall_{A} \exists_{x} A = f[x] \qquad (\textit{Cantor's paradox})$$

As can be seen, our version of the paradox states that there is no surjective function from a type α to the type of sets over α.

Proofs in Theorema-HOL typically begin with the **proof** command for applying a suitable introduction- or elimination rule to the current goal through backchaining. In the case of (*Cantor's paradox*) a suitable rule is *negation introduction*: assume the positive part of the negated formula and derive a contradiction. Hence, the next step in the proof consists of assuming that there exists a surjective function from α to $\mathsf{SET}[\alpha]$:

assume '$\exists_{f} \forall_{A} \exists_{x} A = f[x]$'

Now, since this assumption is an existentially quantified formula, we may introduce a Skolem constant $\overline{f}$ that witnesses it:

then obtain '$\overline{f} :: (\alpha_{\_} \to \mathsf{SET}[\alpha_{\_}])$' **where** "1": '$\forall_{A} \exists_{x} A = \overline{f}[x]$' **by** rule

The name of the Skolem constant is completely arbitrary; in principle, it has nothing to do with the name of the bound variable. We assign the label "1" to the new formula, where the bound variable f is replaced by $\overline{f}$, for being able to refer to it afterward. In the end, however, we have to prove that "1" is indeed a consequence of the known existential formula, which we accomplish by the single command '**by** rule': similar as **proof**, this command applies a suitable introduction- or elimination rule (via backchaining) to the current subgoal, which is *existential elimination* here. The **then** at the beginning of this line merely indicates that for proving "1" we need the previously obtained fact, namely the existential assumption.

Now comes the ingenious part of the proof: the construction of a set which violates "1". As is well known, we simply have to take the set of all x such that $x \notin \overline{f}[x]$, i. e. the set $\mathsf{Collect}[\lambda_{x}\, x \notin \overline{f}[x]]$. Since we are going to use this set several times in the remainder of the proof, we introduce a *syntactic abbreviation* for it:

let $\tilde{X}$ **be** '$\mathsf{Collect}[\lambda_{x}\, x \notin \overline{f}[x]]$'

This command registers an abbreviation for the syntactic variable $\tilde{X}$ which is unfolded already during parsing. Hence, from now on $\tilde{X}$ is a mere shortcut for $\mathsf{Collect}[\lambda_{x}\, x \notin \overline{f}[x]]$.

Having $\tilde{X}$, we can now instantiate the universally quantified formula "1" by it:

[5]We deliberately choose a "simple" example that can be formulated in terms of elementary set theory.

from "1" **have** '$\underset{x}{\exists}\, \tilde{X} = \overline{f}[x]$' ..

have sets up a new subgoal, which in this case is "1" instantiated by $\tilde{X}$. **from** expresses that the subsequent formulas, in this case "1", are needed to prove the subgoal; the proof is again a single application of backchaining wrt. a suitable rule (this time *universal elimination*), invoked by '..'. In fact, '..' is just a shortcut for '**by** rule'.

We are again left with an existentially quantified formula in our knowledge base, so as before we replace the bound variable by a fresh Skolem constant:

then obtain $\overline{x}$ **where** '$\tilde{X} = \overline{f}[\overline{x}]$' ..

Note that as before the new subgoal can be proved by backchaining, invoked by '..'.

From the formula we have just proved, $\tilde{X} = \overline{f}[\overline{x}]$, we now want to obtain the *symmetric variant*, i. e. the equality with the two sides exchanged:

hence "eq": '$\overline{f}[\overline{x}] = \tilde{X}$' **by** (rule "= sym")

hence is just a shortcut for '**then have**'. The new subgoal is once again proved by backchaining, but this time we specify the appropriate rule explicitly by passing it as an argument to 'rule': it is "= sym", the formula expressing symmetry of equality, which is by default available in Theorema-HOL. As can be seen, we assign the label "eq" to the new formula.

Finally, we can derive the desired contradiction by proving that $\overline{x}$ is contained in $\overline{f}[\overline{x}]$ iff it is not contained in that set. So, we set up a new subgoal expressing precisely this fact:

have '$\overline{x} \in \overline{f}[\overline{x}] \Leftrightarrow \overline{x} \notin \overline{f}[\overline{x}]$'

This time, however, the new subgoal cannot be proved by a simple invocation of a pre-defined proof method like backchaining, but we have to develop the proof manually. Therefore, just as at the very beginning of this whole proof, we begin a new subproof by issuing the command **proof**, which again applies a suitable introduction rule to the current subgoal. Since the subgoal is an equivalence, the introduction rule applied is *equivalence introduction*, meaning that we must prove both directions of the equivalence separately. The order in which we attack these two new subgoals is arbitrary; we start with the direction from left to right, i. e. we first assume the left-hand-side of the equivalence:

assume '$\overline{x} \in \overline{f}[\overline{x}]$'

Knowing $\overline{x} \in \overline{f}[\overline{x}]$, we also know that $\overline{x}$ is contained in $\tilde{X}$ by virtue of "eq":

hence '$\overline{x} \in \tilde{X}$' **by** (simp only: "eq")

Here we make use of simplification, invoked by 'simp': rewrite the current subgoal and all explicitly noted facts (i. e. knowledge passed along by **then** or **hence**) wrt. the given equalities (in this case only "eq"). More information about simplification can be found below.

Next, we expand the definition of $\in$:

hence '$\mathsf{Contains}[\tilde{X}, \overline{x}]$' **unfolding** "Element def" .

As can be seen, we do not use simplification but the **unfolding** command: **unfolding** behaves similarly as general simplification, in the sense that it also rewrites the current subgoal and all explicitly noted facts wrt. the given equalities (this time (*Element def*)), but it is less powerful in the sense that it cannot handle conditional equalities. After expanding the definition of $\in$, the goal is identical to the explicitly noted fact, so the proof is finished by assumption (shortcut '.').

Now, all that is left to be done in order to infer the right-hand-side of the original equivalence, $\overline{x} \notin \overline{f}[\overline{x}]$, is again simplification wrt. one of the characteristic properties of $\mathsf{Contains}$ and $\mathsf{Collect}$, namely the fact that $\mathsf{Contains}$ is a right-inverse of $\mathsf{Collect}$, expressed by formula (*Collect inverse*) (recall that $\tilde{X}$ is of the form $\mathsf{Collect}[\ldots]$):

then show '$\overline{x} \notin \overline{f}[\overline{x}]$' **by** (simp add: "Collect inverse")

show is like **have** but does not set up a new subgoal, but instead informs Theorema to prove the current subgoal (typically originating from the previous invocation of **proof**) now. This finishes the $\Rightarrow$-part of the proof of the equivalence.

The other direction proceeds analogously, so we present the full proof at once. Only note that the final **thus** is a mere shortcut for '**then show**':

assume '$\overline{x} \notin \overline{f}[\overline{x}]$'
hence '$\overline{x} \notin \tilde{X}$' **by** (simp only: "eq")
thus '$\overline{x} \in \overline{f}[\overline{x}]$' **by** (simp add: "Element def" "Collect inverse")

This, finally, completes the proof of $\overline{x} \in \overline{f}[\overline{x}] \Leftrightarrow \overline{x} \notin \overline{f}[\overline{x}]$. The proof of (*Cantor's paradox*) is finished by using this equivalence and deriving a contradiction by simplification:

thus 'False' **by** simp

The whole proof is summarized in Figure 2.3. The presentation of the proof suffers from the limitations of a black-and-white paper. In Theorema, for instance, a b. f. constants such as $\overline{f}$ and $\overline{x}$ are typeset in a different font and different color, and so are syntactic variables like $\tilde{X}$. Furthermore, when developing the proof interactively, Theorema sets up a *proof status* window which always displays the *proof situation* (consisting of subgoals and available knowledge) corresponding to the current position of the insertion point in the proof notebook; this behavior was modeled after proof-development in Isabelle/jEdit.

Apart from $\overline{x} \in \overline{f}[\overline{x}] \Leftrightarrow \overline{x} \notin \overline{f}[\overline{x}]$, all subgoals arising in the proof are proved by applying a single proof method through **by**: either by simplification ("simp") or by backchaining ("rule"). These two proof methods are described more thoroughly below.

proof and **by** are similar in that they apply a given proof method to the current subgoal (if no proof method is passed to **proof**, as above, it defaults to "rule", i. e. backchaining). Actually, the only difference between **proof** and **by** is that the latter attempts to prove the subgoal entirely and fails if this is not possible, whereas the former only fails if the given proof method cannot be applied at all. If the proof method can be applied in several ways, which happens frequently with backchaining, **by** automatically backtracks over all possibilities until one leads to success.

One of the most frequently used proof commands is missing in the above proof: **fix** for introducing abf. constants originating from universally quantified

```
**proof**
    **assume** ‘$\underset{f}{\exists}\underset{A}{\forall}\underset{x}{\exists}\ A = f[x]$’
    **then obtain** ‘$\overline{f} :: (\alpha_{\_} \to \mathsf{SET}[\alpha_{\_}])$’ **where** “1”: ‘$\underset{A}{\forall}\underset{x}{\exists}\ A = \overline{f}[x]$’ **by** rule
    **let** $\tilde{X}$ **be** ‘$\mathsf{Collect}[\underset{x}{\lambda}\ x \notin \overline{f}[x]]$’
    **from** “1” **have** ‘$\underset{x}{\exists}\ \tilde{X} = \overline{f}[x]$’ ..
    **then obtain** $\overline{x}$ **where** ‘$\tilde{X} = \overline{f}[\overline{x}]$’ ..
    **hence** “eq”: ‘$\overline{f}[\overline{x}] = \tilde{X}$’ **by** (rule “= sym”)
    **have** ‘$\overline{x} \in \overline{f}[\overline{x}] \Leftrightarrow \overline{x} \notin \overline{f}[\overline{x}]$’
    **proof**
        **assume** ‘$\overline{x} \in \overline{f}[\overline{x}]$’
        **hence** ‘$\overline{x} \in \tilde{X}$’ **by** (simp only: “eq”)
        **hence** ‘$\mathsf{Contains}[\tilde{X}, \overline{x}]$’ **unfolding** “Element def” .
        **then show** ‘$\overline{x} \notin \overline{f}[\overline{x}]$’ **by** (simp add: “Collect inverse”)
    (*next*)
        **assume** ‘$\overline{x} \notin \overline{f}[\overline{x}]$’
        **hence** ‘$\overline{x} \notin \tilde{X}$’ **by** (simp only: “eq”)
        **thus** ‘$\overline{x} \in \overline{f}[\overline{x}]$’ **by** (simp add: “Element def” “Collect inverse”)
    (*qed*)
    **thus** ‘$\mathsf{False}$’ **by** simp
```

Figure 1: Proof of (*Cantor's paradox*) in Theorema-HOL. Note that (*next*) and (*qed*) are mere comments without any formal meaning.

goals. For example, if the formula to be proved is

$$\underset{x,y}{\forall}\ P[x,y]$$

its proof would typically begin with

```
**proof**
    **fix** $\overline{a}$ $\overline{y}$
    **show** ‘$P[\overline{a}, \overline{y}]$’ ⟨*proof*⟩
```

Of course, the names of the abf. constants introduced by **fix** are completely arbitrary; they do not have to agree with the names of the bound variables.

We now give a more thorough account on the two proof methods used in the proof of (*Cantor's paradox*). At the moment, a couple of other proof methods are also implemented in Theorema-HOL, but they are rather specialized and not of such general interest. Implementing further powerful, general-purpose proof methods is future work. Here one must note that proof methods have no effect on the logical correctness of the reasoning system, as they only construct “candidate” proofs which must pass through Theorema-Core's fixed logical kernel.

Simplification. Simplification, or equational rewriting, repeatedly rewrites all subterms of the current subgoal wrt. (universally quantified, conditional) equalities given as arguments to the proof method or appearing among the currently selected knowledge. Equalities are solely used for rewriting from left to right, i. e. instances of the left-hand-side are replaced by the corresponding instances of the right-hand-side. Conditions of equalities are proved by simplification recursively; this distinguishes general simplification from **unfolding**, which can only handle unconditional equalities. Theorema hardly[6] attempts to detect potentially non-terminating or non-confluent sets of rewrite-rules, but leaves this task exclusively to the user.

Simplification is invoked by the "simp" keyword passed as an argument to **by** or **proof**. By default, a (user-extensible) set of standard simplification rules is used for rewriting, but it is possible to overwrite this set by specifying the rules to be used explicitly ("only: ..."), by adding further rules ("add: ..."), or by removing rules ("rem: ..."). The default rule-set contains, for instance, many simplification rules for propositional- and predicate logic.

Backchaining. Backchaining means replacing the current subgoal by zero or more other, hopefully "simpler", subgoals by virtue of a (universally quantified) implication: the consequent of the implication is tried to be unified with the current subgoal, and if a unifier is found, the subgoal is replaced by the corresponding instances of the premises of the implication. If there are no premises (i. e. the "implication" is strictly speaking no implication), no new subgoals emerge and the proof is finished. Since unification in a higher-order setting is of unification type 0, i. e. a unification problem might have infinitely many unifiers none of which is most general, backtracking strategies are employed to iterate over different results; this, in fact, is being taken care of by **by**. As a technical detail please note that the unification procedures currently implemented in the system are *higher-order pre-unification* [10] and, as a special subcase, *higher-order pattern unification* [15].

Backchaining is invoked by the "rule" keyword passed as an argument to **by** or **proof**. Similar to simplification, there is a (user-extensible) default set of backchaining rules that is applied to the current subgoal, containing all the usual introduction- and elimination rules of propositional- and predicate calculus. But in contrast to simplification it is not possible to add further rules, but only to specify the rules to be used explicitly by passing them as additional parameters (as happened with "= sym" in the proof of (*Cantor's paradox*)).

Remark 4. Both simplification and backchaining can only handle rules in Core-format, i. e. with meta-level universal quantifiers, implications and equalities. Simplification, however, automatically converts object-level rules into this format, if necessary.

Remark 5. The current implementation of proving in Theorema-HOL provides the basic infrastructure for automatically generating natural-language proof documents from proofs given by the user in Theorema-HOL's proof language. This makes the mathematical theories formalized in the system accessible to non-experts who are not familiar with the intricacies of the proof language, but who do know how to read mathematical proofs in general (in textbooks, for example). Although the basic infrastructure is available, the actual generation of proof documents has not been implemented yet; this is future work.

[6]Rules that are obviously non-terminating, because their RHS is an instance of their LHS, *are* detected and only applied under certain circumstances.

2.4 Creating Theory Archives

After formalizing a theory, it is desirable to be able to make use of its definitions, lemmas, etc. in other formalizations. In Theorema-HOL this can be achieved easily through the **import** theory command:

import "Sets"

imports the formalization of elementary set theory, stored in the Theorema notebook "Sets.nb" in a directory where Theorema-HOL can find it, by evaluating all of its Theorema-relevant cells (theory commands, formulas, proofs) unless it has been imported already. After this process is finished, all concepts formalized in "Sets" are available in the current theory, and furthermore the dependency of the current theory on "Sets" is recorded internally. Of course, all theories "Sets" itself depends upon are imported as well.

Importing a theory as sketched above, by evaluating all cells in the corresponding notebook, can be very time-consuming, especially if several theories are to be imported. Thus, it is possible to create so-called *theory images*: an image of a theory is just a file containing all formalized concepts of that theory and all its ancestors that can be loaded efficiently by Theorema. **import** always searches for theory images, and in case it finds one, automatically loads the image instead of evaluating the notebook the image originates from. The only drawback of theory images is that, in the current implementation, at most one image can be loaded into a session of Theorema-HOL; once an image has been loaded, all further theories are imported in the "slow" way. The reason for this shortcoming is that otherwise it is difficult (though possible) to rule out contradictory definitions of the same constant in different images. Improving this is future work.

3 Theoretical Background

We now explain the theoretical background of Theorema-Core and Theorema-HOL, and their precise relationship.

3.1 Theorema-Core: The Meta-Logic

Theorema-Core is the meta-logic of Theorema-HOL, in the sense that Theorema-HOL is formulated in the language of Theorema-Core. It implements constructive simply-typed higher-order logic with top-level (or "ML-style") polymorphism [14, 5] by providing a fixed set of primitive inference rules that define the semantics of the two basic type constructors FORM (the truth type) and $\rightarrow$ (the function type) and the four basic concepts λ (abstraction), $\approx$ (extensional equality, of type $\alpha \rightarrow \alpha \rightarrow \mathsf{FORM}$), $\longrightarrow$ (entailment, of type $\mathsf{FORM} \rightarrow \mathsf{FORM} \rightarrow \mathsf{FORM}$) and $\bigwedge$ (universal quantification, of type $(\alpha \rightarrow \mathsf{FORM}) \rightarrow \mathsf{FORM}$). In addition, it also implements a mechanism for ensuring that mathematical theories are developed in a sound and consistent manner, namely either by definitional theory extensions (which are guaranteed to be conservative), or by formally proved theorems, or by arbitrary axioms (which might not be conservative, but are clearly marked so that users who want to build upon a theory can inspect them easily). Hence, Theorema-Core ensures that if the (clearly marked) axioms of a theory are consistent, then the whole theory is consistent, too, and all proved theorems are indeed logical consequences (w. r. t. the inference rules of Theorema-Core) of the axioms; users have no chance to "cheat" but must formalize theories according to the strict rules imposed by Theorema-Core.

As we have noted above, the semantics of λ, $\eqsim$, $\longrightarrow$ and $\bigwedge$ is defined exclusively in terms of the primitive inference rules of Theorema-Core. In the design of these rules we strove for a good trade-off between *simplicity* and *efficiency*: simplicity, because users of the system must trust the soundness of the inferences and the fact that they adequately reflect the intended meaning of λ etc., and efficiency because checking proof candidates wrt. the inferences should be doable in reasonable time. Hence, neither is the resulting set in any sense "minimal", nor are the individual rules formulated in the "simplest" possible way. Instead of listing the rules here, we refer to [13].

Based on this small, trusted kernel of primitive inferences, arbitrary proof methods can be developed: a proof method is a function that takes as input a proof situation P (i. e. a structure consisting, basically, of a goal formula, a list of local assumptions, and some additional (type-) context) and constructs a sequence of primitive inferences and other proof methods which are deemed suitable to make progress in proving P. Since all proof methods ultimately boil down to sequences of primitive inferences, they may be regarded as mere abbreviations of such sequences, and hence their implementation does not affect the soundness of the inference kernel at all. This implies that users of the system may safely develop their own proof methods and add them to the existing arsenal of predefined proof methods that come in conjunction with Theorema-Core, like the powerful simplification- and backchaining methods mentioned in Section 2.3, at any time.

What holds for the interplay between primitive inferences and proof methods also holds for the basic mechanisms for theory extensions and sophisticated tools built upon them. For instance, according to the rules of Theorema-Core, new constants may only be introduced by means of explicit, non-recursive definitions. Based on that, however, it is possible to devise tools that allow users to introduce new constants in quite different ways, e. g. inductively or as recursive functions, by automatically converting their characterizations into proper definitions demanded by Theorema-Core. For example, the characterization of an inductive predicate through its introduction rules can be replaced by a proper definition by virtue of the Knaster-Tarski fixpoint theorem in complete lattices [19]. In a similar vein, though less sophisticated, the definition of the type of sets as an isomorphic copy of the type of boolean-valued functions in Section 2.1 can be accomplished without giving a non-emptiness proof only because there is an internal tool which constructs such non-emptiness proofs automatically when copying whole types. The very basic mechanism for type definitions in Theorema-Core *always* requires a non-emptiness proof.

Summarizing, Theorema-Core consists of two main components:

- a fixed, trusted kernel of primitive inference rules and mechanisms for theory extensions, which jointly ensure sound and consistent formal theories, and

- an extensible, growing collection of proof methods and "theory tools", which enable the effective and efficient use of Theorema-Core (or an object-logic like Theorema-HOL) for actual formalizations.

Based on these two main components, object-logics like Theorema-HOL can be developed easily in the framework of Theorema-Core, as described in the next subsection.

Remark 6. Theorema-Core closely resembles Isabelle/Pure [18], although its implementation in *Mathematica* differs from the implementation of Isabelle/Pure in ML quite a lot; see [13] and Section 5 for details.

3.2 Logical Foundations of Theorema-HOL

Theorema-HOL constitutes the classical counterpart to the constructive logic of Theorema-Core: it is simply-typed classical higher-order logic with extensional equality, Hilbert-choice and top-level polymorphism, formulated within Theorema-Core and hence an object logic of it. We now describe in detail how Theorema-HOL is defined (and how other object logics may be defined in the future). The actual formalization is also part of the 'Core+HOL+Interactive' add-on package (see footnote 1): notebook 'CoreTheories/HOL/HOL.nb'.

All begins with the introduction of the object-level truth type BOOLas the type of formulas of Theorema-HOL, in contrast to the existing meta-level truth type FORM of formulas of Theorema-Core:

typedecl 'BOOL'

registers the new type BOOL without attaching any meaning to it; in particular, at this time BOOL might still contain an arbitrary number of elements (but at least one), although later axioms will ensure that it contains exactly the two elements True and False.

Next, we axiomatically introduce a *truth judgment*, i. e. a function of type BOOL $\rightarrow$ FORM which maps formulas of the object logic to formulas of the meta logic, in order to embed the object logic into the meta logic for being able to reason about the former within the latter. This is achieved by

judgment 'TrueQ::(BOOL $\rightarrow$ FORM)'

which declares the new constant TrueQ to be a function of the given type, and additionally informs the type inference mechanism that from now on, whenever a term of type FORM is expected but a term of type BOOL is found, it shall tacitly insert the coercion TrueQ to ensure well-typedness. In particular, formulas entered by the user into the system as axioms or theorems may now have type BOOL instead of FORM. Moreover, the pretty-printer suppresses all occurrences of TrueQ when printing formulas for the sake of better readability.

Besides the truth judgment, a handful of other constants needs to be introduced axiomatically, too. This time, however, we use the standard **decl** and **abbreviation** commands for declaring fresh constants:

decl 'Equal::($\alpha_{\_} \rightarrow \alpha_{\_} \rightarrow$ BOOL)'
decl 'Implies::(BOOL $\rightarrow$ BOOL $\rightarrow$ BOOL)'
decl 'Some::(($\alpha_{\_} \rightarrow$ BOOL) $\rightarrow \alpha_{\_}$)'
abbreviation 'Iff' $\rightleftharpoons$ 'Equal::(BOOL $\rightarrow$ BOOL $\rightarrow$ BOOL)'

Equal (infix $=$) and Implies (infix $\Rightarrow$) should be self-explanatory, Some represents Hilbert's choice operator, and Iff (infix $\Leftrightarrow$) is a mere abbreviation for equality on type BOOL. The precise meaning of these four constants is specified through the following four axioms, one for each constant:[7]

$$\bigwedge_{s,t::\alpha_{\_}} \mathsf{TrueQ}[s = t] \;\approx\; (s \approx t) \qquad (=\textit{ reflect})$$

$$\bigwedge_{P,Q} \mathsf{TrueQ}[P \Rightarrow Q] \;\approx\; (P \longrightarrow Q) \qquad (\Rightarrow\textit{ reflect})$$

$$\bigwedge_{P,Q} (P \longrightarrow Q) \longrightarrow (Q \longrightarrow P) \longrightarrow P \Leftrightarrow Q \qquad (\Leftrightarrow I)$$

$$\bigwedge_{P::(\alpha_{\_}\to\mathsf{BOOL}),x} P[x] \longrightarrow P[\mathsf{Some}[P]] \qquad (\textit{SomeI})$$

Axioms $(=$ *reflect*$)$ and $(\Rightarrow$ *reflect*$)$ express that $=$ and $\Rightarrow$ are the precise object-level analogues of their meta-level counterparts $\approx$ and $\longrightarrow$, respectively, in the sense that $s = t$ holds iff $s \approx t$ is provable and that $P \Rightarrow Q$ holds iff Q is provable from P. Note that these two axioms explicitly feature the truth judgment TrueQ for the sake of clarity, although the type inference mechanism could automatically insert it if missing, as explained above. The third axiom, $(\Leftrightarrow I)$, states that two terms of type BOOL are equivalent if one follows from the other. The last axiom, $(\textit{SomeI})$, simply expresses that $\mathsf{Some}[P]$ gives some x such that $P[x]$ holds, provided that such an x exists; otherwise, it gives a random, unspecified element of the respective type.

Finally, we can introduce the remaining usual logical connectives and quantifiers by means of ordinary definitions:

$$\mathsf{True} \;:\Leftrightarrow\; \left(\underset{x::\mathsf{BOOL}}{\lambda}\; x = \underset{x}{\lambda}\, x\right) \qquad (\textit{True def})$$

$$\bigwedge_{P,Q} \mathsf{Forall}[P,Q] \;:\Leftrightarrow\; \left(\underset{x}{\lambda}\,(P[x] \Rightarrow Q[x]) = \underset{x}{\lambda}\,\mathsf{True}\right) \qquad (\forall\textit{ def})$$

$$\bigwedge_{P,Q} \mathsf{Exists}[P,Q] \;:\Leftrightarrow\; \left(\underset{R}{\forall}\left(\underset{P[x]}{\forall} Q[x] \Rightarrow R\right) \Rightarrow R\right) \qquad (\exists\textit{ def})$$

$$\mathsf{False} \;:\Leftrightarrow\; \left(\underset{P}{\forall} P\right) \qquad (\textit{False def})$$

$$\bigwedge_{P} \mathsf{Not}[P] \;:\Leftrightarrow\; (P \Rightarrow \mathsf{False}) \qquad (\neg\textit{ def})$$

$$\bigwedge_{P,Q} \mathsf{And}[P,Q] \;:\Leftrightarrow\; \left(\underset{R}{\forall}\,(P \Rightarrow Q \Rightarrow R) \Rightarrow R\right) \qquad (\wedge\textit{ def})$$

$$\bigwedge_{P,Q} \mathsf{Or}[P,Q] \;:\Leftrightarrow\; \left(\underset{R}{\forall}\,(P \Rightarrow R) \Rightarrow (Q \Rightarrow R) \Rightarrow R\right) \qquad (\vee\textit{ def})$$

$$\bigwedge_{P,x,y} \mathsf{If}[P,x,y] \;:=\; \mathsf{Some}[\underset{z}{\lambda}\;(P \Rightarrow z = x) \wedge (\neg P \Rightarrow z = y)] \qquad (\textit{If def})$$

Some remarks about these definitions are in order:

[7]Recall from Section 3.1 that $\bigwedge$ is the meta-level universal quantifier.

- By default, Forall, Exists, Not, And and Or are automatically assigned their common mathematical syntax $\forall$, $\exists$, $\neg$, $\wedge$ and $\vee$, respectively. This is because of the very names of these constants, to which Theorema (and *Mathematica*) associates that special syntax.

- $\forall$ and $\exists$ have two arguments, because Theorema typically distinguishes between the condition and the main part of a quantified formula. For instance, it is common to write something like $\underset{x \in A}{\forall} x > 2$ where x is the bound variable, $x \in A$ is the condition and $x > 2$ is the main part of the quantified formula. Internally, this expression translates into $\mathsf{Forall}[\underset{x}{\lambda}\, x \in A, \underset{x}{\lambda}\, x > 2]$. Obviously, it is up to the human user to split a quantified formula into condition and main part just as he or she pleases. Furthermore, it is also possible to omit the condition altogether on input, in which case it defaults to $\underset{x}{\lambda}\, \mathsf{True}$.

Remark 7. Theorema-Core only allows definitions of new constants of the form $c \approx \ldots$, but the above definitions clearly do not match this pattern. Instead, there are meta-universally quantified variables, the left-hand-sides are constants applied to arguments, and the main relation symbols are not $\approx$ but $:\Leftrightarrow$. Still, Theorema-Core accepts the definitions, because Theorema-HOL is equipped with a tool that automatically and tacitly converts definitions of the above kind into definitions following the required $c \approx \ldots$ pattern. This is accomplished by replacing $:\Leftrightarrow$ and $:=$ by $\approx$, and by adding new λ-abstractions on the right-hand-side for each of the universally quantified variables. For instance, ($\neg$ *def*) then becomes $\mathsf{Not} \approx \left(\underset{P}{\lambda}\, P \Rightarrow \mathsf{False}\right)$. In fact, the same also happens with (*Element def*) in Section 2.2.

From the above axioms and definitions it is possible to derive all the well-known properties of the various logical constants, including the usual introduction- and elimination rules, simplification rules (e. g. the De Morgan's laws), etc., and further concepts, like unique existence and the description operator, can easily be defined in terms of the existing ones.

Remark 8. The logic of Theorema-HOL is classical, although this is not enforced by a dedicated axiom. Instead, the law of the excluded middle follows from the Axiom of Choice (which is (*SomeI*) in our setting, since it asserts the existence of a choice function for every type) by Diaconescu's Theorem [7].

We conclude this section by sketching an alternative approach to formalizing higher-order logic, and why we did not follow it. Namely, instead of embedding classical higher-order logic as an object logic into the constructive logic of Theorema-Core by introducing a separate truth type (BOOL) and a corresponding truth judgment (TrueQ), we could instead have extended Theorema-Core *itself* by the law of the excluded middle to make its logic classical. Then, the truth type would still be FORM, and there would be no need for neither a truth judgment nor for object-level equality, implication and universal quantification. The reason why we did not pursue this seemingly simpler approach is *predicativity*: the law of the excluded middle (or any axiom equivalent to it) would refer to all formulas of type FORM then – and be a formula of this type itself, meaning that it would refer to itself. For reasons well known, self-reference in axiomatizations is not desirable, since it might easily lead to inconsistency of the theory, and therefore we (once again) followed the example of Isabelle/HOL and formalized classical higher-order logic in the way outlined in this section. Predicativity is achieved since none of the axioms quantifies variables whose types involve FORM.[8] See also page 33 of [17] for a short discussion of predicativity in the context of the Isabelle proof assistant.

4 Comparison to Standard-Theorema

In this section we summarize and comment on the main similarities and differences between Theorema-HOL and standard Theorema 2.0 [4].

4.1 Similarities

The common ground of Theorema-HOL and standard-Theorema is the underlying *Mathematica* system and, in particular, the sophisticated graphical user interface built upon it [22]. Indeed, although internally Theorema-HOL and standard-Theorema are implemented quite differently (see Section 4.2), on the outside these differences are scarcely visible: formulas are input and output in the same way,[9] with all kinds of two-dimensional syntax and notation, and the so-called *Theorema commander* window [22] is used in both incarnations of Theorema to interact with the system in an intuitive and user-friendly manner. Furthermore, both systems provide a mechanism for automatically creating human-readable proof documents from (automatically or interactively generated) proofs.

Besides that, there are not many further similarities – although Theorema-Core (and thus Theorema-HOL) is an add-on package for Theorema 2.0. This is because Theorema-Core only uses the graphical user interface provided by standard-Theorema, but implements most functionality concerned with reasoning and theory exploration itself.

4.2 Differences

Most differences between Theorema-HOL and standard-Theorema have been mentioned in the previous sections already; we summarize them here again for the sake of completeness:

- Theorema-Core (and thus Theorema-HOL) enforce a clear separation between the fixed logical kernel on the one hand, and extensible, user-definable proof methods on the other hand. Standard-Theorema is much more flexible in this regard, allowing users to implement their own primitive rules of inference.

- In Theorema-HOL theories may only be developed by certain well-defined means for achieving consistency. For instance, new constants can only be introduced by non-overlapping non-recursive definitions (and other, more sophisticated tools which internally, however, rely on such definitions as

[8]Even the type variables appearing in these axioms are not allowed to be instantiated by types involving FORM.

[9]There are some minor differences, like *type annotations*, though.

well). Although arbitrary (and hence potentially inconsistent) axiomatizations are allowed, too, they are explicitly marked as such and can easily be inspected by users of the respective theories, without having to search the whole theories manually for axioms.

- Theorema-Core and Theorema-HOL are typed logics for ruling out paradoxes such as Russell's. Standard-Theorema, on the other hand, is completely untyped.
- Theorema-HOL relies on the concept of theory commands for users to interact with the system; examples are **decl** for declaring fresh constants and **typedef** for defining new types (see Section 2). Standard-Theorema does not utilize theory commands at all.
- Theorema-HOL focuses on interactive proving rather than automatic proof search, as standard-Theorema does and has been doing from its very origins. This, however, is not inherent to Theorema-HOL, but merely an intentional, pragmatic design decision, since automatic proof search in higher-order logic is an utmost difficult problem in practice and, of course, even undecidable in theory. Nevertheless, Theorema-HOL provides a range of sophisticated proof methods that are able to prove sufficiently simple goals automatically; as an example recall the simplifier presented in Section 2.3.

There is another important difference between Theorema-Core and standard-Theorema that has not been mentioned so far: the internal representation of terms and formulas. Standard-Theorema adopts a very flexible representation of formulas, allowing symbols of arbitrary arity (even flexible arity), Currying, and several kinds of variable binders; furthermore, it stores bound variables nominally. Theorema-Core on the other hand, is much more restrictive: every function symbol is applied to at most one argument and λ is the only binder-construct; Currying is used to model application of functions to more than one arguments, as in, say, $f[x_1][x_2]$, and de Bruijn indices [6] are used to efficiently store bound variables.[10]

Remark 9. The internal representation of terms and formulas in Theorema-Core does not affect their input and output at all. For instance, it is still perfectly fine to enter terms like $f[x_1, x_2, x_3]$, which are automatically converted into $f[x_1][x_2][x_3]$ during parsing and converted back into their un-Curried form during pretty-printing.

5 Comparison to Isabelle

In the previous sections we have already pointed out various similarities between Theorema-HOL and Isabelle/HOL, which we summarize in this section. Additionally, we also list the main differences between our system and Isabelle. We assume some familiarity with Isabelle here.

[10]But of course the original names of bound variables are also stored for pretty-printing.

5.1 Summary of Similarities

The main similarities between Theorema-HOL (and Theorema-Core) and Isabelle are the following:

- The meta logic of Theorema-HOL, i. e. Theorema-Core, is simply-typed constructive higher-order logic with top-level polymorphism and extensional equality. This parallels the situation in Isabelle.
- Theorema-HOL is an object logic of Theorema-Core and equipped with its own truth type, connectives and quantifiers, just as Isabelle/HOL is an object logic of Isabelle/Pure.
- Theorema-Core has a fixed, small logical kernel implementing the primitive rules of inference, and a collection of sophisticated user-extensible proof methods and theory tools built upon it. This design ensures both the soundness of reasoning and the consistency of formal theories.
- The proof- and theory languages of Theorema-HOL bear a close resemblance to Isabelle/Isar, after which they were modeled.

5.2 Different Features

Type Classes and Context Management. Isabelle/HOL supports *type classes* [9] and sophisticated context management through so-called *locales* [1]. In a nutshell, type classes are an elegant means to build up hierarchies of mathematical domains in a structured manner, similar to *functors* that have been proposed for Theorema [21]. In fact, a type class is just a collection of types, each defining a fixed set of operations (a "signature") and satisfying a fixed set of axioms; for instance, Isabelle/HOL provides type classes for orderings, additive- and multiplicative groups, rings, fields, any many more. Locales, on the other hand, allow the user to set up local theory contexts characterized by local parameters and assumptions, in which new constants may be introduced and theorems may be stated; all such constants and theorems are tacitly parameterized over the parameters of the locale, and all theorems are additionally constrained by the assumptions of the locale.

As of yet, Theorema-HOL lacks mechanisms for effective context management like type classes and locales, but instead asks the user to take care of a systematic build-up of his theories himself.

Type Definitions. Type definitions in Theorema-HOL differ from those in Isabelle/HOL in that they are are based on predicates rather than sets. Indeed, looking at the definition of the type $\mathsf{FSET}[\alpha]$ of finite sets in Section 2.1 one sees that the type is characterized by the predicate $\mathsf{isFinite}$. Type definitions in Isabelle/HOL, however, expect sets as the characterization of new types, as, for instance, $\{x.\ \mathsf{isFinite}[x]\}$.

Of course, the parallels of predicates and sets being obvious and well-known, there is scarcely any difference between the two kinds of type definitions in Theorema-HOL and Isabelle/HOL from the practical point of view. But Isabelle/HOL requiring *sets* in type definitions implies that the type of sets itself cannot be introduced through a type definition there! Instead, the type of sets must be introduced manually by means of declarations of the type and the respective type morphisms and axioms describing the characteristic properties of said morphisms. In Theorema-HOL, on the other hand, the type $\mathsf{SET}[\alpha]$ of sets can easily and conveniently be introduced by a simple type definition, as illustrated in Section 2.1.

Interactive Proving. Although the proof language of Theorema-HOL was modeled after Isabelle/Isar and, hence, features the same commands and keywords with similar meanings, there are three commands whose behavior in Theorema-HOL slightly deviates from the behavior of their analogues in Isabelle/Isar: **fix**, **assume** and **show**. In Isabelle, these commands are *passive*, in the sense that they do not directly manipulate the current proof situation, but rather construct an implicit theorem which *afterwards* is employed to prove the current goal. For example, if the current proof situation contains the goal G and the local assumptions A and B, then it is perfectly fine to prove it by writing something like

assume B
show G $\langle proof \rangle$

because the goal $A \Longrightarrow B \Longrightarrow G$ is indeed a simple consequence of the implicitly constructed theorem $B \Longrightarrow G$. This offers the human author of proofs quite some flexibility, since unused assumptions (like A in the above example) can be omitted entirely in proofs, and furthermore is the order in which assumptions (**assume**) and constants (**fix**) are introduced by the user completely independent of their order in the current proof situation. Unfortunately, this flexibility has some negative side effects, too. Namely, whether a theorem constructed by a sequence of **fix**, **assume** and **show** is indeed suitable for proving the current goal can be decided only at the very end (when issuing the final **show**). So, it may happen to users who make some assumptions using **assume** and prove some intermediate results using **have**, to be in the end informed by Isabelle that their assumptions are too strong to prove the current goal – and this can be fairly annoying.

In Theorema-HOL, said three commands actively participate in the reduction of the current proof situation. **assume**, for instance, instantly checks whether the given assumptions are indeed premises of the current goal, in which case these premises are stripped off and stored as local facts of the proof context instead; if they are no premises of the goal, the user is informed about this fact immediately and the application of the proof command fails. The downside of this setup is that authors of proofs are not as flexible in omitting unused assumptions and regarding the order in which a. b. f. constants are introduced as they are in Isabelle/Isar.

User Interface. Last, but not least, one of the most distinctive features of Theorema (not only of our add-ons Theorema-HOL and Theorema-Core) is its appealing and intuitive graphical user interface based on *Mathematica*: formulas can be input and are displayed in conventional notation and two-dimensional syntax (variables under binders, fractions whose numerator is displayed above the denominator, matrix-like arrangements, etc.), formal content can be decorated with informal explanatory text, figures and diagrams, and proofs can be presented in a form intelligible not only to expert users of the system but even to ordinary mathematicians ignorant of any particular formal proof language.[11] Since a lot has been written about Theorema's capabilities in this respect, we refer the interested reader for more information to the literature, e., g. [22, 4].

Isabelle, on the other hand, keeps a clear separation between the development of formal theories in an editor (like jEdit, which is a text editor capable of some very restricted form of two-dimensional syntax) and their dissemination as TEX documents or HTML pages (which can be generated automatically). Hence, Isabelle lacks the seamless integration of development and dissemination Theorema has ever since laid its focus on.

5.3 Differences in the Implementation

Primitive Rules of Inference. The primitive rules of inference of Theorema-Core differ from those of Isabelle/Pure in two respects. First and foremost, there are more rules, and more powerful ones, in Theorema-Core than in Isabelle/Pure, although they give rise to the same deducibility relation. This is because in Theorema-Core we strove for a good trade-off between simplicity and efficiency of the inference kernel: simplicity meaning that the rules, though not as simple and few in number as theoretically possible, should nevertheless be easy to comprehend by humans to gain trust in their adequately characterizing the logic of Theorema-Core, and efficiency meaning that the rules can be used efficiently by Theorema for checking proofs.

Second, the inference rules are implemented quite differently in our system compared to Isabelle. In Isabelle, following the well-established LCF approach to theorem proving [8], the primitive rules of inference are implemented as constructors of a dedicated type `thm` for theorems in ML. Then, ML's static type-checking mechanism ensures that every term of type `thm` was constructed only using its constructors (i. e. the inference rules) and hence is indeed a valid theorem. So, checking the validity of propositions in Isabelle is reduced to type-checking in Isabelle's implementation language ML. In Theorema-Core, we had to pursue a different approach, since the implementation language of Theorema, i. e. *Mathematica*, lacks type-checking mechanisms altogether. Instead, we manually implemented a function for checking proofs wrt. the inference rules, which resides in the trusted kernel of the system, too. We refer to [13] for a more detailed description of the implementation of Theorema-Core's proof-checking functionality.

Higher-Order Unification. The driving force behind Isabelle's reasoning machinery is higher-order (pre-)unification: several inference rules in the trusted kernel rely on it, and it therefore must be implemented in the trusted kernel, too. Higher-order unification, however, is a difficult problem, and efficient implementations of procedures for solving this problem are fairly difficult to comprehend by non-experts who want to convince themselves of their correctness—and their correctness *is* crucial for the soundness of the entire logical kernel of Isabelle!

Matters are different in Theorema-Core: there, higher-order unification is *not* part of the trusted kernel, but implemented in a separate component whose correctness does not affect the soundness of the deduction system at all. This

[11]But keep in mind Remark 5.

is because no single primitive inference rule relies on higher-order unification whatsoever; instead, higher-order unification is only used by certain high-level proof methods (e. g. backchaining) for creating suitable instances of universally quantified formulas.

Axiomatization of Theorema-HOL. The axiomatization of Theorema-HOL presented in Section 3.2, consisting of the three axioms ($=$ *reflect*), ($\Rightarrow$ *reflect*) and (*SomeI*), differs from the axiomatization of Isabelle/HOL. There, only (*SomeI*) is an axiom, but the other two formulas are theorems derived from other axioms characterizing the semantics of equality and implication. The reason for our deviating from Isabelle/HOL's axiomatization is that we strove for a system of axioms as small and as concise as possible. For comparison, Isabelle/HOL has ten axioms in total (not counting the Axiom of Infinity). But of course both axiomatizations are equivalent to each other.

6 Conclusion

Theorema-HOL is a relatively young effort, and so there are still many things to do before it can effectively be used in large-scale formalizations of mathematics. This does not only concern the development of further basic theories, like numbers, lists, algebraic structures, etc., but also the implementation of more sophisticated and powerful proof methods and (semi-)automatic means for augmenting theories, e. g. by inductive definitions, algebraic data types and recursive functions.

References

[1] Clemens Ballarin. Tutorial to Locales and Locale Interpretation. In Laureano Lambán, Ana Romero, and Julio Rubio, editors, *Contribuciones Científicas en Honor de Mirian Andrés Gómez*, pages 123–140. Servicio de Publicaciones de la Universidad de La Rioja, 2010. Part of the Isabelle documentation, `https://isabelle.in.tum.de/dist/Isabelle2016/doc/locales.pdf`.

[2] Grzegorz Bancerek, Czeslaw Byliński, Adam Grabowski, Artur Korniłowicz, Roman Matuszewski, Adam Naumowicz, Karol Pak, and Josef Urban. Mizar: State-of-the-art and Beyond. In Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, and Volker Sorge, editors, *Intelligent Computer Mathematics (Proceedings of CICM'15)*, volume 9150 of *LNAI*, pages 261–279. Springer, 2015.

[3] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[4] Bruno Buchberger, Tudor Jebelean, Temur Kutsia, Alexander Maletzky, and Wolfgang Windsteiger. Theorema 2.0: Computer-Assisted Natural-Style Mathematics. *Journal of Formalized Reasoning*, 9(1):149–185, 2016.

[5] Luca Cardelli. Basic Polymorphic Typechecking. *Science of Computer Programming*, 8(2):147–172, 1987.

[6] Nicolaas G. de Bruijn. Lambda Calculus Notation with Nameless Dummies: A Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[7] Radu Diaconescu. Axiom of Choice and Complementation. *Proceedings of the American Mathematical Society*, 51:176–178, 1975.

[8] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer, 1979.

[9] Florian Haftmann. *Haskell-style Type Classes with Isabelle/Isar*, 2017. Part of the Isabelle documentation, `https://isabelle.in.tum.de/dist/Isabelle2017/doc/classes.pdf`.

[10] Gérard Huet. Résolution d'équations dans les langages d'ordre $1, 2, \ldots, \omega$, 1976. Thèse d'état, Université Paris 7, Paris, France.

[11] Ondřej Kunčar and Andrei Popescu. Comprehending Isabelle/HOL's Consistency. In H. Yang, editor, *Programming Languages and Systems (ESOP 2017)*, volume 10201 of *LNCS*, pages 724–749. Springer, 2017.

[12] Ondřej Kunčar. Correctness of Isabelle's Cyclicity Checker: Implementability of Overloading in Proof Assistants. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, pages 85–94. ACM, 2015.

[13] Alexander Maletzky. A New Reasoning Framework for Theorema 2.0. Technical report, RISC, Johannes Kepler University Linz, 2017. `http://www.risc.jku.at/publications/download/risc_5461/Paper.pdf`, presented as work in progress at CICM'2017 (Edinburgh, UK, July 17–21).

[14] Robin Milner. A Theory of Type Polymorphism in Programming. *J. Computer and System Sciences*, 17(3):348–375, 1978.

[15] Tobias Nipkow. Functional Unification of Higher-Order Patterns. In Moshe Vardi, editor, *Eighth Annual Symposium on Logic in Computer Science*, pages 64–74. IEEE Computer Society Press, 1993.

[16] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[17] Lawrence C. Paulson. The Foundation of a Generic Theorem Prover. Technical Report UCAM-CL-TR-130, University of Cambridge, 1988. `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-130.pdf`.

[18] Lawrence C. Paulson. Isabelle: The next 700 Theorem Provers. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[19] Alfred Tarski. A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[20] Makarius Wenzel. *The Isabelle/Isar Reference Manual*, 2017. Part of the Isabelle documentation, `https://isabelle.in.tum.de/dist/Isabelle2017/doc/isar-ref.pdf`.

[21] Wolfgang Windsteiger. Building Up Hierarchical Mathematical Domains Using Functors in THEOREMA. In Alessandro Armando and Tudor Jebelean, editors, *Proceedings of Calculemus'99, Trento, Italy*, volume 23 of *Electronic Notes in Theoretical Computer Science*, pages 401–419. Elsevier, 1999.

[22] Wolfgang Windsteiger. Theorema 2.0: A Graphical User Interface for a Mathematical Assistant System. In Cezary Kaliszyk and Christoph Lueth, editors, *UITP'2012*, volume 118 of *EPTCS*, pages 72–82. Open Publishing Association, 2012.

[23] Wolfram Research, Inc. *Mathematica*. `http://www.wolfram.com/mathematica/`.