Theorem Proving for the MMT System

Luca Wolff
Advisor: Florian Rabe
Friedrich-Alexander-Universität Erlangen-Nürnberg

April 22, 2022

Abstract Isabelle's Sledgehammer shows how powerful ATPs can be in assisting when formalizing theories and proofs. This thesis reports on the development of an extension (using the existing Extension/Plugin system) for the MMT System that can translate MMT theories to TPTP and then uses the LEO-III ATP to try to automatically prove accordingly tagged conjectures in theories. The translation process determines the used logic in MMT and matches them with the available logics in TPTP. When successful it then pattern matches the terms and outputs the according TPTP code step by step. While it has its current use cases, it's especially a potent foundation for evolution of the current features and addition of new ones.

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Luca Wolff

Contents

1 Introduction

1.1 Motivation

Formalized knowledge is a valuable resource for automated systems which can parse and process the knowledge for a multitude of purposes: Extracting specific information, generating derivative facts and formally proving theories or programs are just a few ways how formalized knowledge can be used to assist humans in research, development, education and more. This field is getting more popular and research achieved some important milestones. The Flyspeck project's goal was creating a formal proof for the Kepler Conjecture, whose original proof from 1998 consists of about 300 pages of text and 40000 lines of code. It is so complex that the team of referees still wasn't completely certain about the correctness of the proof after a 5 year review period. The circumstances made it an attractive target for formalizing to settle doubts and further popularize formal methods in mathematics. The proof of the Odd Order Theorem by Feit and Thompson, was fully formalized in Coq. What makes it special is that, other than typically formalized proofs, the also very lengthy written proof was meant to be fully read and understood and didn't rely on heavy computational effort. The key of this formalization was in developing a framework, which allowed for simple and efficient switching of views for the same mathematical objects, as the proof featured peculiar combinations of theories. Other advancement made, were a formally verified compiler and micro-kernel, which provide safety on a lower level than ever before. [Tea03] [KU14] [GAA+13] [KEH+09] [Ler06] The MMT system tries to abstract further than logical frameworks do, which already are an important tool for a more efficient workflow and experimenting with logics and their applications. So the MMT system's greater generality allows for developing other logical frameworks with it. But its rather young age makes it lack in formalized content compared to the usual applications using logical frameworks, like for example the proof assistant Isabelle. Currently all proof in MMT are written by hand and writing proofs by hand is already a tedious task for relatively trivial theories. But when you want to be sure to not introduce mistakes in essential building blocks of further theories a formal proof is very important. Automated Theory Provers (ATPs) are a powerful tool, which allow proving of many theories without human intervention. So to make work with systems that allow for formalizing theories in Logic, like MMT, more efficient and secure, as humans are also more prone to error, it makes sense to introduce ATPs to MMT. Being able to resort to the big selection of existing ATPs would empower MMT and boost further progress in formalization. [Rab18] [isa]

1.2 Related Work

Most notable of similar works has been Sledgehammer for the Isabelle proof assistant. While there were earlier attempts at combining interactive and automatic theorem proving, Sledgehammer is still used regularly by users and can solve a good amount of nontrivial goals. Sledgehammer translates goals of Is-

abelle scripts to TPTP and uses ATPs in the background to generate the proof of the goal, ready to be inserted into the original Isabelle script. The biggest design goal was to make the whole process invokable by users with only one click and require no other manual steps to be really viable for productive usage. [MQP06] [PB10]

1.3 Objectives

The objectives of this work include:

Seamless Integration The automated proving process should be easily accessible from the usual workflow with MMT and should not interrupt it

Keeping Code Idiomatic The MMT code containing the theory that is supposed to be proven, should require only minimal changes and not hinder the usage of practical MMT features for code re-usage and structuring, like for example Includes and Constants.

Conservative Add-On This extension of MMT should not require any changes of the core implementation, as it should not be necessary to achieve the goals aimed at and just would introduce a danger of breaking the trusted, long-standing inner workings of the MMT System.

Proof Term Generation Proofs for the same unchanged theories should be reproducible. But because there is the chance of ATPs interfering with this exact objective by bugs or just new features, as ATPs are highly sensitive programs, there should be measures in place to at least be able to reconstruct the latest successful proof. This could be done by logging output of the ATP or translating the proof back to MMT code, which would conserve the proof in an optimal manner, allowing reading and re-usage in further work.

1.4 Contribution

To be able to communicate shortcomings of the translation directly in the resulting file, comments could be used, but Leo-III's TPTP parser, which was used by the implementation to construct TPTP in code, didn't support comments. As part of this thesis I implemented[1] this feature. This included different kinds of comments' representations in code, how they get output and parsing them from TPTP files, to make it a proper implementation I can pull-request to the original project on Github. This succeeded, which makes it a supported feature and allows for updating this dependency in the future of this thesis' project.

2 Preliminaries

The standard logics for ATPs are FOL, SFOL and Typed HOL, we will shortly introduce those and then their syntax in TPTP and MMT. To better highlight the important semantic details and to deprive attention from syntactical details, simplified versions of the original languages MMT and TPTP are used. They

[1] `https://github.com/leoprover/scala-tptp-parser/pull/9`

also abstract from technical details of the implementations and resulting peculiarities. For better illustration and to showcase practicality we work with the Peano Axioms as an running example throughout this thesis. We start at how we would express them in the respective Languages and eventually step through the translation process from MMT to TPTP by this example.

2.1 Base Languages

We introduce the supported logics and introduce the representable parts of the axioms respectively.

2.1.1 FOL

First-**O**rder **L**ogic is an extension of propositional logic. It allows for quantification over objects.

Example 2.1 (Peano Axioms)**.**

$$\begin{aligned} &\textbf{III.} \quad \forall x(0 \neq S(x)) \\ &\textbf{IV.} \quad \forall x \forall y(S(x) = S(y) \Rightarrow x = y) \end{aligned} \tag{1}$$

As induction(**V**) required quantification over a function it is not representable in first-order logic. Also **I** and **II** aren't expressible without types.

2.1.2 SFOL

Sorted **F**irst-**O**rder **L**ogic extends FOL by adding types to variables. Quantification happens over objects of a specific type, instead of the whole range of objects.

Example 2.2 (Peano Axioms)**.**

$$\begin{aligned} &\textbf{I.} \quad 0 \in \mathbb{N} \\ &\textbf{II.} \quad \forall x \forall y((x \in \mathbb{N} \wedge S(x) = y) \Rightarrow y \in \mathbb{N}) \end{aligned} \tag{2}$$

With the addition of types we can now express axiom **I** and **II**.

2.1.3 Typed HOL

In extension of quantification over objects **H**igher-**O**rder **L**ogic allows Quantification over properties. Therefore Typed HOL also offers types for properties — propositions and functions — and enables for example expressing assertions over all properties of a single type of object.

Example 2.3 (Peano Axioms)**.**

$$\textbf{V.} \forall X \colon \mathbb{N} \to \underbrace{o}_{\text{boolean}} \ (X(0) \wedge \forall n \colon \mathbb{N}(X(n) \Rightarrow X(S(n)) \Rightarrow \forall x \colon \mathbb{N}(X(x)))) \tag{3}$$

And finally we can also introduce induction(**V**) with quantification over the predicate X representing a set.

2.2 Representation in MMT

The MMT system allows formalizing of declarative languages, like for example different types of logics and mathematical theories, in a foundation-independent manner. This enables high reusability of algorithms and structures through generic implementations. To further empower users of MMT in creating expressive theories, MMT ships the **urtheories** archive. (Archives are basically libraries of MMT Theories) It includes the base theories for MMT's **L**ogical **F**ramework, which provides a lot of tools for expressing any logic related content. These general logical concepts are further concretized in the **LATIN2** archive, which includes logical symbols, like quantifiers and connectives for logics from propositional logic to higher-order logic in a multitude of variations with different extensions and specifics. It also adds the rules of natural deduction, which allows to express complete proofs in the different logics and even let them to be typechecked. For our purposes we don't need to get further into the details of the structure of **LATIN2**. Everything we need from the mentioned archives will be defined in the following. [MMT] [LAT]

Definition 2.1 (MMT Theory)**.** Theories in MMT are defined as:

$$Theory = Declaration^* \tag{4}$$

Definition 2.2 (MMT Declaration)**.** Declarations can declare other Declarations to be included from a different MMT file or declare named information themselves. Additionally Declarations have types in SFOL and HOL, but cannot in FOL. Their formal definition is:

$$\begin{aligned} Declaration = {} & \text{include } \boldsymbol{ident} \\ & \mid Constant \end{aligned} \tag{5}$$

$$Declaration = \boldsymbol{ident} \colon Type = \underbrace{\zeta}_{\text{Definition}} \tag{6}$$

$$\begin{aligned} Type &= (T \rightarrow)^* \quad \underbrace{T}_{\text{Return Type}} \\ T &= \texttt{tm}\ LogicType \\ &\mid \texttt{tp} \\ &\mid \texttt{prop} && \text{or also } \texttt{bool} \text{ in HOL} \\ &\mid \vdash \zeta \\ &\mid \texttt{term} && \text{Type of terms in FOL} \\ LogicType &= (\boldsymbol{ident}\ \rightarrow_{\texttt{tm}})^*\ \boldsymbol{ident} \end{aligned} \tag{7}$$

Definition 2.3 (MMT Formula)**.** Formulas in MMT are defined as the follow-

ing:

$$
\begin{array}{lll}
\zeta, \theta = \zeta \wedge \theta & & \\
\quad \mid \zeta \vee \theta & & \\
\quad \mid \neg\zeta & & \\
\quad \mid \zeta \Rightarrow \theta & & \\
\quad \mid \forall[\boldsymbol{var}]\ \zeta & & \\
\quad \mid \exists[\boldsymbol{var}]\ \zeta & & \\
\quad \mid \boldsymbol{ident} \underbrace{\zeta^*}_{\text{arguments}} & \text{Atomic/Function/Predicate} & \\
\quad \mid \zeta = \theta & \text{Equation} & \\
\quad \mid \zeta \iff \theta & \text{Equivalence} & \\
\quad \mid \zeta @ \theta & \text{Lambda Application} & \\
\quad \mid \lambda[\boldsymbol{var}]\ \zeta & \text{Lambda Definition} &
\end{array}
\tag{8}
$$

Definition 2.4 (Variable and Identifier in MMT)**.**

$$
\begin{array}{ll}
\boldsymbol{var} = \boldsymbol{ident} \colon \texttt{tm}\ \mathit{LogicType} & \text{represents a variable of a quantor} \\
\boldsymbol{ident} \text{ is just an alphanumeric identifier in the usual sense} &
\end{array}
\tag{9}
$$

Example 2.4 (Peano Axioms)**.**

$$
\begin{array}{lll}
& N \colon tp & \text{This type declaration is necessary for the axioms.} \\
\textbf{I.} & \mathit{zero} \colon \texttt{tm}\ N & \\
\textbf{II.} & \mathit{successor} \colon \texttt{tm}\ (N \to N) & \text{This type deceleration is required by MMT,} \\
 & & \text{but also implicitly conveys axiom \textbf{II}.} \\
\textbf{III.} & \mathit{no\_confusion} \colon \vdash \forall[n : \texttt{tm}\ N] \neg(\mathit{successor} @ n = \mathit{zero}) & \\
\textbf{IV.} & \mathit{injectivity} \colon \vdash \forall[n \colon \texttt{tm}\ N] \forall[m \colon \texttt{tm}\ N]((\mathit{successor} @ n = \mathit{successor} @ m) \Rightarrow (n = m)) & \\
\textbf{V.} & \mathit{induction} \colon \vdash \forall[X \colon \texttt{tm}\ (N \to \mathit{bool})](X @ \mathit{zero} \wedge \forall[n \colon \texttt{tm}\ N](X @ n \Rightarrow X @ (\mathit{successor} @ n)) & \\
 & \quad \Rightarrow \forall[x \colon \texttt{tm}\ N] X @ x) &
\end{array}
\tag{10}
$$

2.3 Representation in TPTP

Thousands of **P**roblems for **T**heorem **P**rovers is foremost a library of problems designed for testing ATPs. These test problems are formulated in the TPTP language, which is specifically designed for being a very general input language for a wide range of different Provers. Today most ATPs do understand TPTP as a language for problem input. This makes it a very attractive choice as a general interface between a flexible system working with logics and theories with only manual proofs and a multitude of ATPs which have different specializations, like which logics they support and tactics they use.

Note that we're introducing a simplified version of TPTP here, which conveys the used functional features, with less syntactical overhead to ease explanation and formal definitions.

Definition 2.5 (TPTP File)**.** A TPTP File consists out of a list of Deceleration. In this project we use the file endings ".ax" for axiom files and ".p" for problem files as the official TPTP Problem Library does. These kinds of files don't work any differently, they just indicate their use case. We define File as:

$$File = Decl^* \tag{11}$$

Definition 2.6 (TPTP Declaration)**.** TPTP Declarations either declare something new depending on their so-called formula role[2] or include declarations from another file. We represent them like this:

$$\begin{aligned} Decl = \ & \text{NewType}(\tau) \\ & \mid \text{NewTypedObject}(\boldsymbol{ident}, \tau) \\ & \mid \text{AxiomDecl}(\phi) \\ & \mid \text{DefinitionDecl}(\text{Forall}(\boldsymbol{var}^*, \\ & \qquad \text{Equal}(\text{Atomic}(\boldsymbol{ident}, \phi^*), \psi) \\ & \quad \mid \text{Equiv}(\text{Atomic}(\boldsymbol{ident}, \phi^*), \psi)) \\ & \mid \text{ConjectureDecl}(\phi) \\ & \mid \text{Include}(\boldsymbol{ident}) \end{aligned} \tag{12}$$

The DefinitionDecl is a subset of the AxiomDecl, but we still introduce it here for readability. It is modeled after a formula with the 'definition' role mentioned in the TPTP grammar itself. ConjectureDecl is equivalent to AxiomDecl besides being tagged as the conjecture and thus should only appear a single time. The ATP will try to prove the conjecture ϕ.

$$\begin{aligned} AtomicType = \ & \text{ConstantType}(\boldsymbol{ident}) \mid DefinedType \\ & \text{For readability we only write the } \boldsymbol{ident} \text{ of ConstantType} \end{aligned} \tag{13}$$

$$\begin{aligned} DefinedType = \ & \$\text{o} && \text{represents boolean - true and false} \\ & \mid \$\text{i} && \text{represents non-empty individuals/objects} \\ & \mid \$tType && \text{represents type of all types} \\ & \mid \$real \mid \$rat \mid \$int && \text{represents } \mathbb{R}, \mathbb{Q} \text{ and } \mathbb{N} \text{ respectively} \end{aligned} \tag{14}$$

A constant type is one which was declared in a TypeDecl beforehand using \$tType as the internal supertype, which represents the type of all types. The other defined types can be used to build propositions $T > \$oType$ and functions

[2] Taken from the official syntax document.

$T > \$iType$.

$$\begin{aligned} \tau_{\text{tff}} &= \text{MappingType}(\text{UnitaryType}, \text{AtomicType}) \\ &\mid \mathit{AtomicType} \\ \text{UnitaryType} &= \text{ProductType}(\mathit{AtomicType}^*) \\ &\mid \mathit{AtomicType} \end{aligned} \tag{15}$$

τ_{tff} is very limited in expressiveness compared to τ_{thf}, because higher-order types are only allowed on top-level, which means we can declare functions which take in FOL objects – a single one of atomic type or multiple combined into a product type – as arguments and return a single FOL object. No complex types can be used on application level.

$$\begin{aligned} \tau_{\text{thf}} &= \text{MappingType}(\tau_{\text{thf}}, \tau_{\text{thf}}) \\ &\mid \text{ProductType}(\tau_{\text{thf}}, \tau_{\text{thf}}) \\ &\mid \text{UnionType}(\tau_{\text{thf}}, \tau_{\text{thf}}) \\ &\mid \mathit{AtomicType} \end{aligned} \tag{16}$$

For τ_{thf} everything is allowed, we can take and return complex-typed arguments, like functions, product and even unions types in opposition to the strict rules of TFF.

Definition 2.7 (TPTP Formula)**.** Formulas in TPTP are defined as the following:

$$\begin{aligned} \phi, \psi = {} & \text{And}(\phi, \psi) \\ & \mid \text{Or}(\phi, \psi) \\ & \mid \text{Not}(\phi) \\ & \mid \text{Impl}(\phi, \psi) \\ & \mid \text{Forall}(\boldsymbol{var}^*, \phi) \\ & \mid \text{Exists}(\boldsymbol{var}^*, \phi) \\ & \mid \text{Atomic}(\boldsymbol{ident}, \underbrace{\phi^*}_{\text{arguments}}) \quad || \quad \text{Atomic}_{\text{fof}}(\boldsymbol{ident}, \underbrace{\text{Atomic}(\_, \_)^*}_{\text{arguments}}) \\ & \quad \text{For readability we only write the } \boldsymbol{ident} \text{ if there are no arguments} \\ & \mid \text{Equal}(\phi, \psi) \\ & \mid \text{Equiv}(\phi, \psi) \\ & \mid \text{Apply}(\phi, \psi) \qquad \text{Only in SFOL \& HOL} \\ & \mid \text{Lambda}(\boldsymbol{var}^*, \phi) \qquad \text{Only in SFOL \& HOL} \end{aligned} \tag{17}$$

Definition 2.8 (Variable and Identifier in TPTP)**.**

$$\begin{aligned} &\boldsymbol{var} = \text{NewTypedObject}(\boldsymbol{ident}, \tau) \qquad \text{represents a variable of a quantor} \\ &\boldsymbol{ident} \text{ is just an alphanumeric identifier in the usual sense} \end{aligned} \tag{18}$$

3 Translating MMT Conjectures to TPTP

3.1 Usage/Initiation of the Process

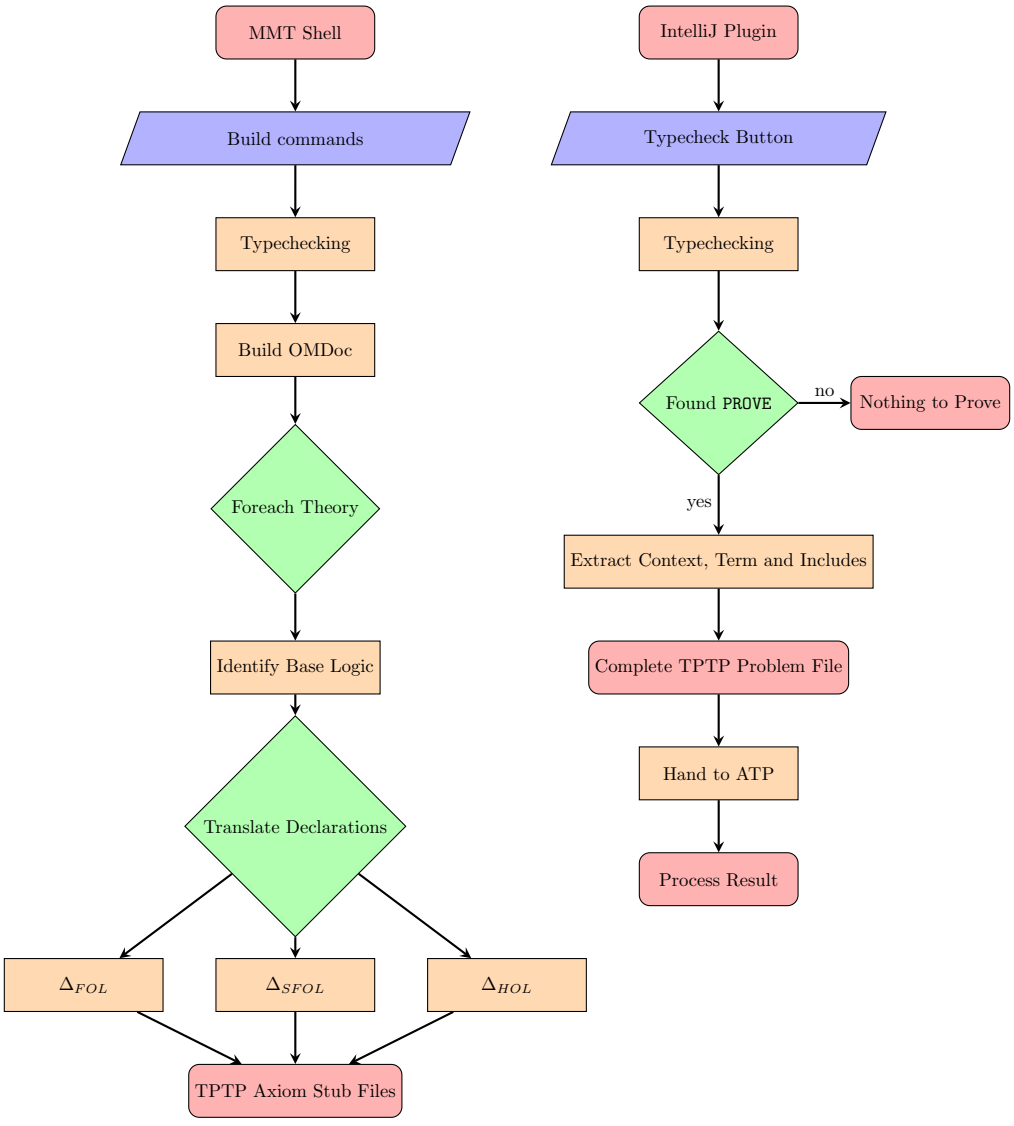

Figure 1: Flowchart of the Translating/Proving Process

The most basic way to kick off the translation process is to use the MMT Shell. Here no ATP is involved and we exclusively translate a whole MMT file into

several TPTP Axiom files, one for each theory. So to continue our example it would like something like this:

```
> build MMT/LATIN2 mmt-omdoc peano.mmt
> build MMT/LATIN2 tptp peano.omdoc
```

We create an OMDoc file which conveys the whole structure of the original MMT file in an easily processable manner. So we use that OMDoc file then to start the process of translating to TPTP. Translation runs separately for each theory of the source MMT file, which first needs to determines what logic it is dealing with by going through the implicit – that means fully recursively resolved – imports and checking for the corresponding paths of FOL, SFOL and HOL base theory. In our case we use all 3 base theories, so HOL is selected as it can represent the lower logics as well. After deciding which logic the theory is residing in it calls the specialized translation module of that logic. This split into separate module is needed to account for the large differences in TPTPs grammar, especially for those of untyped and typed logic. In this thesis we simplified TPTP's grammar to make it possible to have a translation function, which requires only a few exceptions for the alternative logics to reduces the complexity and improve readability. For the next step each deceleration will be translated with the translation function Δ which will be formally defined in the next section. After translating the theory line by line the result will be exported into a TPTP Axiom file. Notice that includes will be handled differently, explained in Section 3.4.1. So at this point we have a `peano.ax` file containing the 5 axioms described by TPTP HOL Declarations. A more straightforward approach of triggering the translation process with the goal of actually proving a theory with help of an automated theorem prover is to use the `PROVE` rule, specifically implemented for this purpose. Instead of manually translating a whole MMT file of theories, we just need to insert the `PROVE` rule in the definition of the term we want to prove. In our example we might add a conjecture that a natural number can only ever be equal to zero or be the successor of another number. So we add this to our MMT code:

$$conjecture\colon \vdash \forall[x\colon \texttt{tm}\ N](x = zero) \vee (\exists[y\colon \texttt{tm}\ N] successor@y = x)\ =\ \textbf{PROVE} \tag{19}$$

Invoking the type-checker now triggers an automatism when `PROVE` is found, which takes care of translating theories to TPTP Axiom files plus putting the Problem file together which contains the tagged conjecture. This is then handed to the ATP, which in our case is Leo-III.

[Koh06]

3.2 Mapping MMT's LF onto TPTP's Available Formulae

TPTP support and differentiates between 6 different kinds of formulae:
TPI: **T**PTP **P**rocess **I**nstruction is not like the other formulae, it is not representing a logic, but rather used for communicating with the ATP it is run with. It is worth mentioning here, as it allows for finer control over how the problem is

read and handled by the prover. For modelling complicated functions of MMT, which exceed the boundaries of the usual (following) logics present in TPTP, sending special commands to manipulate groups of logical formulae could prove beneficial.
CNF: **C**onjunctive **N**ormal **F**orm is self explaining.
FOF: **F**irst-**O**rder **F**orm represents First-Order Logic formulae.
TCF: **T**yped first-order **C**lausal **F**orm represents Sorted/Typed CNF.
TFF: **T**yped **F**irst-order **F**orm represents Sorted/Typed First-Order Logic formulae.
THF: **T**yped **H**igher-order **F**orm represents Sorted/Typed Higher-Order Logic formulae.

We are going to focus on FOF, TFF, THF as they are the most common and they are the most powerful in their group. CNF is a subgroup of FOF, which is mostly used, because algorithms can work with them well and not because they are nice to write or read. So most First-Order theories will be in a more complex form anyway. The same applies to TCF. Note alse that both of these forms are neither supported by Leo-III's parser nor the ATP itself.

Now we need to decide which logic to translate the MMT code, which is using the very broad LF. Luckily we can can look at the imports of the theory and defer more information about what specfic part of LF is used.

From the meta theory we can see, that EXAMPLEPATH is included. To figure out which logic this is coming from we need to follow the included path.

When we stumble over a theory or maybe just a single term, which depends on higher or different logics which TPTP does not support, we have multiple options. We can just abort the whole process and optimally tell the user which specific part is not translatable, so not automatically provable. The user then can try to adjust the problem to a supported form or simplify the problem based on external knowledge. Another way of dealing with this situation could be trying to isolate the smallest unrepresentable term and then assume it is truthyness. That way we can still assist the user as much as possible and it only leaves a rest to prove by hand or other measures.

3.3 Definitions of translation

Definition 3.1 (Translating Formulas)**.**

$$
\begin{aligned}
&tr(\zeta \wedge \theta) = \mathrm{And}(tr(\zeta), tr(\theta))\\
&tr(\zeta \vee \theta) = \mathrm{Or}(tr(\zeta), tr(\theta))\\
&tr(\neg\zeta) = \mathrm{Not}(tr(\zeta))\\
&tr(\zeta \implies \theta) = \mathrm{Impl}(tr(\zeta), tr(\theta))\\
&tr(\zeta = \theta) = \mathrm{Equal}(tr(\zeta), tr(\theta))\\
&tr(\zeta \iff \theta) = \mathrm{Equiv}(tr(\zeta), tr(\theta))\\
&tr(\underbrace{\boldsymbol{ident}}_{\text{name of atom/function/predicate}} \underbrace{\zeta_1, \ldots, \zeta_n}_{\text{arguments}}) = \mathrm{Atomic}(c, tr(\zeta_1), \ldots, tr(\zeta_n)) \quad \zeta_i \text{ is a 0-ary Atomic in FOL}\\
&tr_\tau(\forall[\boldsymbol{var}]\underbrace{\zeta}_{\text{body}}) = \mathrm{Forall}(\mathit{convar}(\boldsymbol{var}), tr_\tau(\zeta))\\
&tr_\tau(\exists[\boldsymbol{var}]\underbrace{\zeta}_{\text{body}}) = \mathrm{Exists}(\mathit{convar}(\boldsymbol{var}), tr_\tau(\zeta))\\
&tr_{HOL}(\lambda[\boldsymbol{var}]\underbrace{\zeta}_{\text{body}}) = \mathrm{Lambda}(\mathit{convar}(\boldsymbol{var}), tr_{HOL}(\zeta))\\
&tr_{HOL}(\zeta@\theta) = \mathrm{Apply}(tr(\zeta), tr(\theta))
\end{aligned}
\tag{20}
$$

Definition 3.2 (Translating Declarations)**.**

$$
\begin{aligned}
&\Delta(\text{include X}) = \mathrm{Include}_{\mathrm{TPTP}}(\mathrm{X}_{\mathrm{raw}}) \text{ where } \mathrm{X}_{\mathrm{raw}} \text{ is X without its includes}\\
&\Delta(T\colon \texttt{tp}) = \mathrm{TPTP}(\text{“type\_”} + T, T, \$\mathit{type})\\
&\Delta(T\colon \texttt{tp} = S) = \Delta(S\colon tp)(\boldsymbol{CurrentlyNotSupported})\\
&\Delta(f\colon \texttt{tm}\ T_1 \to \ldots \to \texttt{tm}\ T_n \to \texttt{tm}\ T) = \mathrm{TPTP}(\text{“type\_”} + f, \mathrm{type}, \mathit{convar}(f\colon \texttt{tm}\ T_1 \to \ldots \to \texttt{tm}\ T_n \to \texttt{tm}\ T)\\
&\Delta(f\colon \texttt{tm}\ T_1 \to \ldots \to \texttt{tm}\ T_n \to \texttt{tm}\ T = \lambda[A_1\colon tmT_1] \ldots \lambda[A_n\colon \texttt{tm}\ T_n]\zeta) =\\
&\qquad = \Delta(f\colon \texttt{tm}\ T_1 \to \ldots \to \texttt{tm}\ T_n \to \texttt{tm}\ T);\\
&\qquad \mathrm{TPTP}(\text{“def\_”} + f, \mathrm{axiom},\\
&\qquad \mathrm{Forall}((\mathit{convar}(A_1\colon T_1), \ldots, \mathit{convar}(A_n\colon T_n)), \mathrm{Equal}(f@A_1@\ldots@A_n, tr(\zeta)))\\
&\Delta(p\colon \texttt{tm}\ T_1 \to \ldots \to \texttt{tm}\ T_n \to \texttt{prop}) = \mathrm{TPTP}(\text{“type\_”} + f, \mathrm{type}, \mathit{convar}(p\colon \texttt{tm}\ T_1 \to \ldots \to \texttt{tm}\ T_n \to \texttt{prop}))\\
&\Delta(p\colon \texttt{tm}\ T_1 \to \ldots \to \texttt{tm}\ T_n \to \texttt{prop} = \lambda\, A_1\colon \texttt{tm}\ T_1.\ \ldots\ \lambda\, A_n\colon \texttt{tm}\ T_n.\ \ t) =\\
&\qquad \Delta(p\colon \texttt{tm}\ T_1 \to \ldots \to \texttt{tm}\ T_n \to \texttt{prop}); \mathrm{TPTP}(\text{“def\_”} + p, \mathrm{axiom},\\
&\qquad \mathrm{Forall}((A_1\colon T_1, \ldots, A_n\colon T_n), \mathrm{Equiv}(p@A_1@\ldots@A_n, tr(t))))\\
&\Delta(ax\colon \vdash F) = \mathrm{TPTP}(ax, \mathrm{axiom}, tr(F))\\
&\Delta(thm\colon \vdash F = pf) = \Delta(thm\colon \vdash F)
\end{aligned}
\tag{21}
$$

Definition 3.3 (Convert Variables)**.** A conversion for a variable in MMT $\boldsymbol{var}_{MMT}$ to a variable in TPTP $\boldsymbol{var}_{TPTP}$ also needs to be defined for the

translation functions.

$$convar(\boldsymbol{var}_{MMT}) = convar(\boldsymbol{ident}\colon Type) = \text{NewTypedObject}(\boldsymbol{ident}, conType(Type))$$

$$\begin{aligned}
&conType(T) = conT(T)\\
&conType(T_0 \to T) = \text{MappingType}(conT(T_0), conT(T))\\
&conType(T_1 \to \cdots \to T_n \to T) =\\
&\quad \text{MappingType}(\text{ProductType}(\text{conT}(\text{T}_1), \ldots, \text{conT}(\text{T}_\text{n})), \text{conT}(\text{T}))
\end{aligned}$$

$$\begin{aligned}
&conT(\texttt{tm}\ LogicType) = conLT(LogicType)\\
&conT(\texttt{tp})\ \boldsymbol{is\ unsupported}\\
&conT(\texttt{prop}) = \$\text{o}\\
&conT(\vdash \zeta)\ \boldsymbol{is\ unsupported}\\
&conT(\texttt{term}) = \$\text{i}
\end{aligned}$$

$$conLT(\boldsymbol{ident}) = \text{ConstantType}(\boldsymbol{ident}) \tag{22}$$

Example 3.1 (Peano Axioms)**.**

$$\Delta(N\colon tp) = \text{TPTP}(\text{“type\_”} + N, N, \$type) \tag{23}$$

The type definition of N is trivially translated in a single step.

I. $$\Delta(zero\colon \texttt{tm}\ N) = \text{TPTP}(\text{“type\_”} + zero, \text{type}, \text{NewTypedObject}(zero, N)) \tag{24}$$

Then we translate the definition of zero, which requires the first usage of the *convar* function.

II. $$\begin{aligned}&\Delta(successor\colon \texttt{tm}\ (N \to N)) =\\ &\quad = \text{TPTP}(\text{“type\_”} + successor, \text{type}, \text{NewTypedObject}(successor, \text{MappingType}(N, N)))\end{aligned} \tag{25}$$

The *successor* function's type gets encoded with a MappingType from N to N.

III. $$\begin{aligned}&\Delta(no\_confusion\colon\ \vdash \forall[n : \texttt{tm}\ N]\neg(successor@n = zero)) =\\ &\quad = \text{TPTP}(no\_confusion, \text{axiom}, \text{Forall}(\text{NewTypedObject}(n, \text{ContantType}(N)),\\ &\qquad \text{Not}(\text{Equiv}(\text{Apply}(successor, n), zero))\\ &\quad))\end{aligned} \tag{26}$$

For the $no\_confusion$ axiom, Δ matches for axiom declaration first and then

calls tr, which in turn constructs the nested TPTP formulas.

IV. $\Delta(injectivity\colon \vdash \forall[n\colon \texttt{tm}\ N]\forall[m\colon \texttt{tm}\ N]((successor@n = successor@m) \Rightarrow (n = m))) =$
$= \text{TPTP}(injectivity, \text{axiom}, \text{Forall}(\text{NewTypedObject}(n, \text{ContantType}(N)),$
$\text{Forall}(\text{NewTypedObject}(m, \text{ContantType}(N)),$
$\text{Impl}(\text{Equal}(\text{Apply}(successor, n), \text{Apply}(successor, m)), \text{Equal}(n, m))$
$)$
$))$
(27)

injectivity is translated similarly, it has two Foralls to declare variables n and m, which are then used in the implication.

V. $\Delta(induction\colon \vdash \forall[X\colon \texttt{tm}\ (N \to bool)]$
$(X@zero \wedge \forall[n\colon \texttt{tm}\ N](X@n \Rightarrow X@(successor@n)) \Rightarrow \forall[x\colon \texttt{tm}\ N]X@x)) =$
$= \text{TPTP}(induction, \text{axiom}, \text{Forall}(\text{NewTypedObject}(X, \text{MappingType}(N, \$o)),$
$\text{Impl}(\text{And}(\text{Apply}(X, zero), \text{Forall}(\text{NewTypedObject}(n, \text{ContantType}(N)),$
$\text{Impl}(\text{Apply}(X, n), \text{Apply}(X, \text{Apply}(successor, n))))),$
$\text{Forall}(\text{NewTypedObject}(x, \text{ContantType}(X)), \text{Apply}(X, x))$
$)$
$))$
(28)

induction is more complex overall, and features a typed object with a MappingType for quantification over functions, but its translation follows the same scheme.

3.4 Handling Structural Features

3.4.1 Includes

Includes in MMT and TPTP differ in functionality. The obvious difference lies in MMT importing theories, while TPTP just imports other files with TPTP code. So to mirror functionality in that regard a single MMT theory should be converted to a single TPTP file. The more grave difference in functionality is how duplicate includes are handled. Duplicates are no rarity, you look no further than EXAMPLE to include multiple theories, which themselves each depend on the same theory THEORY. And MMT can handle this fine, as code of the included theory is not just copy-pasted into the depending file. TPTP on the other hand seemed to do exactly that, after a naive attempt to just swap out the different languages' includes. The specification says files are recursively included and multiple formulae with the same name in a file constitute an error, so that is what we need to work around.

3.4.2 Defined Constants

There can be definitions to functions, proposition in MMT, which describe their functionality with other functions and properties, like a common programming

language would. And then there are definitions to axioms, which contain their proof. TPTP does not support definitions like that, so we have to work around that. There are three obvious ways to handle this. The first would be to just ignore the definition and let the symbol stand on its own. For function and predicates that usually will not work out, as an important part of information – how they exactly work – is left out, however for axiom symbols this is a valid choice, as we do not care how the proof of an axiom looks like, the type checker already made sure it is a valid proof and we can just use it as a normal axiom. Option 2 would be to translate the whole function/proposition decleration and translate it into two seperate lines of TPTP. One containing the type definition, the other containing a statement about how it is behaving. And translating that directly from the definition is actually quite easy. We chose this option, as you can see in Definition 3.2 l.5. The other option would have been to just ignore the decleration completly, but to replace every usage after by the definition, so definition-expand it, which would have been also correct, but is a lot more work and a lot more unpleasing to read, as it gets harder to see the orginial structure of the theory, which probably was thought-out to be easily followable.

3.4.3 Missing Logical Connectives

The logics in LF support quite a few more logical connectives than TPTP does. But many of them are just syntactic sugar for a combination of connectives. They are implemented to make code more readable, as these are very commonly used and widely understood. A good example of this would be the unique existential quantifier:

$$\exists! x \phi(x) \iff \exists x(\phi(x) \wedge \forall y(\phi(y) \to y = x)) \tag{29}$$

So desugaring missing connective in TPTP by replacing them according to these equivalences can resolve this problem.

4 Implementation

The translator produces an TPTP problem represented by data structures of LEO-III's own parser library, which means the problem could be directly forwarded to LEO-III loaded as a library. However, this creates a multitude of problems. The TPTP code references files with its includes, regardless of the main "file" not existing besides the representation in RAM. So to let an ATP try to solve one of our problems, which is not confined to a single "file", we need to save included theories as files on the disk. And the disadvantage of having them on the disk is required disk space, and – much more impactful – having to load each file to memory and parse them, which could hurt performance noticable, when trying to prove a lot of things at once. But having the files in place also has a heap of advantages, we automatically ensure we do not build theories twice, as we can just look up if a file already is in place and additionally can use that as a mechanism to skip re-proving already proven theories completly.

Additionally we now have regular TPTP files at our hands we can use with all the tools that work on them. More concretely, we could use different ATPs which are specialized for our theory subject, which makes this tool a lot more powerful, just like why it was planned to use TPTP in the first place.

To preserve a responsive user experience, starting a proof should not stop the user from continuing with their work, but they still might want to know if their MMT code type-checks as soon as possible. So the ATP process initiated by type-checking should not affect the users writing process - the IDE should not freeze. Plugins of IDEs should be run asynchronous anyways, so that should not be our problem. But it is our decision how to handle the (a)synchronicity of type-checking and proving, and there is an advantage in waiting for the proving process, before finishing the type-check. Sure, the user knows how the results of an failed type-check looks like and it would not be bad to have a similar result for (at least for the ATP) unprovable theories. But there is another possible advantage, the results of the ATP process could be used in further type-checking. So depending on the proof different choices in further type-checking could be made. For now decoupling all processes seemed like the best idea, so the ATP is called asynchronously by type checking and to prevent slowing down a less powerful computer, we have a switch to disable automatic proving on type-checking.

Another way to optimize the performance of proving would be to isolate the `PROVE` in the deceleration the user is currently editing. When there are multiple in the same deceleration a choice has to be made, for now we start a ATP process for each `PROVE` in the current deceleration. All other declarations that are marked for proving are considered as axiomatically true, the user should check if earlier declarations are valid before. To have a final check of validity the option of proving whole theories should be there.

To circumvent the problems mentioned in Includes the processed theory and its included theories are stripped from there includes and declarations after the `PROVE` which initiated the process.

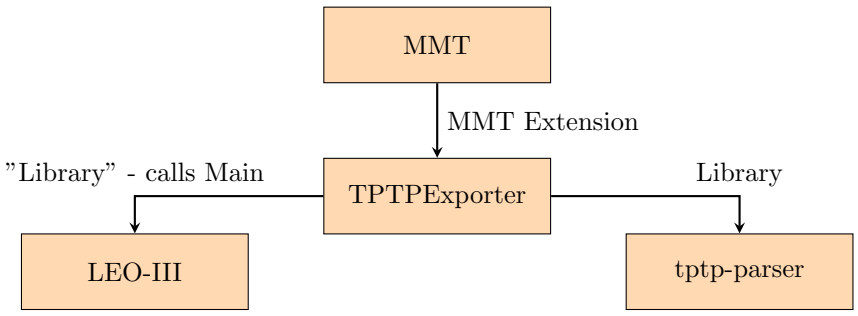

Figure 2: Architecture Diagram

4.1 Extending MMT

We define the class for our MMT extension, with key and extension of the exported files:

```
class TPTPExporter extends StructurePresenter with
    AutomatedProver \{
   override def apply(e : StructuralElement, standalone:
       Boolean = false)(implicit rh : RenderingHandler):
       Unit = {}

   /** a string identifying this build target, used for
       parsing commands, logging, error messages */
   override def key: String = "tptp"

   override val outExt: String = "ax"
```

This is the function which is called by MMT when it find a `PROVE` term in a definition.

```
override def apply(pu: ProvingUnit, rules: RuleSet, levels:
    Int): (Boolean, Option[Term]) = {
   val mod = MPath(pu.component.get.parent.toTriple._1.get,
       pu.component.get.parent.toTriple._2.get)

   val problem_path = combineStubs(mod, pu.context,
       pu.tp)(this.controller) match {
      case Some(problem) => exportProblem(problem, mod)
          //Problem returned
      case None => return (true, None) //Couldn't translate
          problem
   }

   val result = callInternalATP(problem_path)
   (result._1, result._2.map(proof =>
       UnknownTerm(OMSemiFormal(Text("tptp", proof)))))
       //transform result and wrap proof in MMT Terms
}
```

4.2 Translate Function

As an example we take a look at the `translate_formula` function from the `HOLExporter`:

```
def translate_formula(t: Term): THF.Formula = t match {
   case Lambda(v, ty, body) => THF.QuantifiedFormula(THF.^,
       Seq(("V_" + v.toPath, translate_formula(ty))),
       translate_formula(body))
   case simplambda(_, _, f) => translate_formula(f)
```

```
case simpapply(_, _, f, x) =>
    translate_formula(ApplySpine(f, x))
case SimpleFunctionTypes.simpfun(a, b) =>
    funty_builder(List(translate_formula(a)),
    translate_formula(b))

case InternalPropositions.bool.term =>
    THF.FunctionTerm("$o", Nil)

case tforall((ty, Lambda(v, _, body))) =>
THF.QuantifiedFormula(
THF.!,
Seq(
("V_" + v.toPath, translate_formula(ty))
),
translate_formula(body)
)
case texists((ty, Lambda(v, _, body))) =>
THF.QuantifiedFormula(
THF.?,
Seq(
("V_" + v.toPath, translate_formula(ty))
),
translate_formula(body)
)
case tforall(ty, body) =>
val varname =
    Context.pickFresh(body.allVars.map(VarDecl(_)),
    LocalName("x"))._1
translate_formula(tforall(ty, Lambda(varname, ty,
    ApplySpine(body, OMV(varname)))))
case texists(ty, body) =>
val varname =
    Context.pickFresh(body.allVars.map(VarDecl(_)),
    LocalName("x"))._1
translate_formula(texists(ty, Lambda(varname, ty,
    ApplySpine(body, OMV(varname)))))

case and(left, right) =>
THF.BinaryFormula(THF.&, translate_formula(left),
    translate_formula(right))
case or(left, right) =>
THF.BinaryFormula(THF.|, translate_formula(left),
    translate_formula(right))
case impl(left, right) =>
THF.BinaryFormula(THF.Impl, translate_formula(left),
    translate_formula(right))
case equiv(left, right) =>
THF.BinaryFormula(THF.<=>, translate_formula(left),
    translate_formula(right))
```

```
    case tequal(ty, left, right) => {
       THF.BinaryFormula(THF.Eq, translate_formula(left),
           translate_formula(right))
    }
    case notequal(ty, left, right) => {
       THF.BinaryFormula(THF.Neq, translate_formula(left),
           translate_formula(right))
    }
    case not(arg) =>
    THF.UnaryFormula(THF.~, translate_formula(arg))

    case Truth._true(()) =>
    THF.FunctionTerm("$true", Nil)
    case Falsity._false(()) =>
    THF.FunctionTerm("$false", Nil)

    case OMID(f) =>
    THF.FunctionTerm("t_" + f.name.toString, Nil)
    case OMV(x) =>
    THF.Variable("V_" + x.toString)

    case ApplySpine(f, args) =>
        args.map(translate_formula).foldLeft(translate_formula(f))((g,
        arg) => THF.BinaryFormula(THF.App, g, arg))

    case default =>
    currentFormulaComments +:= Comment(CommentFormat.LINE,
        CommentType.NORMAL, "Unknown term/op: " + default)
    THF.FunctionTerm("$true", Nil)
}
```

4.3 Integrating Leo-III

The simplest approach to integrating an ATP is just to start a new external process with the corresponding executable of the ATP, in our case we exclusively worked with Leo-III. There are a few big downsides to using external processes though:

Latency It takes an disproportionate amount of time to start an external process, relative to a direct function call inside of the JVM. To start a new process our program needs to talk to the OS via a system call, which already is relatively expensive, and then the OS needs to spin up the process, allocate resources for it, wait for its full execution. Note that in our case, this isn't just a small native program, but another `jar` which needs another JVM instance to run, which has a significant startup time itself.

Portability An Leo-III executable would need to be provided, while integrating it directly into our extension requires no further configuration.

Flexibility The optimal would be to use Leo-III as a library, which gladly

is easily doable with `jar`s in the JVM. But Leo-III isn't really intended to be used as a library, or rather didn't have such usage in mind when written. So all the neat features of libraries, like exposed functions which you can easily call on a problem and get an `Option[Proof]` back, don't exist. Instead it's all very intertwined with the process of loading and parsing the TPTP files, which also makes it hard to extract these kind of functions. I thought about trying to adapt my fork of Leo-III for easy library usage, but in the end deemed it too time-consuming and wasn't sure if I could do it in a manner which the original author would like to merge into the official repository, which would have been my goal. Because of that I decided to work with Leo-III how it is and just used the entrypoint function `main`. Due to it being a Command-Line-Interface application it takes in arguments via a string and communicates its output over `stdout` and `stderr`, so I had to work with that:

```
def callInternalATP(path: String): (Boolean,
    Option[String]) = {
   val baos = new ByteArrayOutputStream
   val printStream = new PrintStream(baos)
   val err = System.err
   //Leo-III uses System.err for errors and Console for
       other output, which are from Java and Scala
       respectively and so we need to use both APIs to
       capture the output and set the output back to default
       afterwards.
   System.setErr(printStream)
   Console.withOut(printStream) {
      //Call Leo-III's main function with the path to the
          problem file and the "-p" flag to make it output a
          proof.
      leo.Main.main(Array(path, "-p"))
   }
   System.setErr(err)
   //parseResult just finds the Line indicating if proving
       was successfull and if it was additionally outputs
       the TPTP string of the proof
   parseResult(baos.toString)
}
```

So in the end I didn't manage to overcome the flexibility issue, but latency and portability should be much better this way.

4.4 Caching Proofs

To prevent repeated attempts at proving the same conjectures automatically I put a caching mechanism in place. LEO-III can be run with the "-p" flag, which outputs a refutation proof, if one was found. As the proof could be a useful resource for the user and even possibly be translated back to MMT in the future, I decided to integrate this functionality into the caching mechanism.

```
//check if proof cached
val proof_path =
    getOutFileForModule(mod).get.setExtension("proof.tptp")
if (proof_path.exists()) {
   val br = new BufferedReader(new FileReader(proof_path))
   val first_line = br.readLine
   if (first_line == metadata_line(problem_path)) { //here
       we check for equivalent hashes
      println("Proof to '" + mod + "' cached. Skipping..")
      val proof =
          Iterator.continually(br.readLine()).takeWhile(_ !=
          null).mkString
      return (true,
          Some(UnknownTerm(OMSemiFormal(Text("tptp",
          proof)))))
   }
}
```

Here we try to read the first line of the "proof.tptp" file, which is the ending I decided to use for the cached proofs. The first line contains metadata which allows us to check if the following proof is corresponding to the current version of the problem. By a 1-to-1 mapping through practically-collision-free hashing, we can check if the current proof is still valid or if the problem changed and the proving process has to start from scratch.

```
def metadata_line(problem_path: String): String = {
   val buffer = new Array[Byte](8192)
   val md = MessageDigest.getInstance("SHA-256") //We use
       SHA-256, a widely used cryptographic hash function
   val dis = new DigestInputStream(new
       FileInputStream(problem_path), md)
   try { while (dis.read(buffer) != -1) { } } finally {
       dis.close() }
   "% " + Base64.getEncoder.encodeToString(md.digest) //We
       output a comment with the hash in base 64
}
```

When we have just proven a problem, we save a new "proof.tptp" file or update the existing one like this:

```
//Caching proof
result._2 match { //result is a tuple of a boolean
    indicating success and the proof as an Option[String]
   case Some(proof) => outputTo(proof_path) {
      rh(metadata_line(problem_path) + "\n" + proof) //here
          we're adding the metadata line for later validation
   }
}
```

```
namespace latin:/casestudies/2022-tptp

theory Peano : latin:/?HOLND =
    N : tp
    successor : tm (N → N)

    zero : tm N

    //No Confusion Axiom
    axiom3 : ⊢ ∀[n: tm N] ¬(successor @ n ≐ zero)

    //Injectivity Axiom
    axiom4 : ⊢ ∀[n: tm N] ∀[m: tm N] ((successor @ n ≐ successor @ m) ⇒ (n ≐ m))

    //Induction Axiom
    axiom5 : ⊢ ∀[X: tm N → bool] X @ zero ∧ (∀[n : tm N] X @ n ⇒ X @ (successor @ n)) ⇒ ∀[x: tm N] X @ x

    // Conjecture
    conj : ⊢ ∀[x: tm N] (x ≐ zero) ∨ (∃[y: tm N] successor @ y ≐ x) = PROVE
```

Figure 3: Fully Type-Checked Example in MMT

5 Conclusion & Future Work

5.1 Summary

All together I implemented translation functions for MMT theories in First-Order-Logic, Sorted-First-Order-Logic and Higher-Order-Logic to the representation of TPTP from `tptp-parser` from the LEO-III project. I integrated the translation into an MMT Extension, where the Exporter functionality is used to export TPTP axiom files. Additionally on type-checking the `PROVE` term, a TPTP problem file is generated which then will be send to the integrated LEO-III jar, which tries to find a proof. Possible success will be reported to the MMT type-checker and the proof will be cached, so it doesn't need to be computed again on every type-check. For better traceability and easier debugging I added Comment parsing and printing support to `tptp-parser`.

The different semantics of includes in MMT and TPTP required particular attention, to find a solution which was correct and still not to complicated to work with. Also a good understanding of the logical systems helped with writing MMT for test purposes and understanding the data structures especially for TPTP.

5.2 Future Work

For the future there are a lot of things which could be improved or built upon this foundation:

LEO-III Library A library interface for the LEO-III prover would allow

for improved performance, cleaner code and simpler implementation of new features, but it would require some major refactoring and it should be determined if the maintainer is interested in that big of a change.

Proof to MMT Translating the proofs from LEO-III back to MMT would integrate the process of using ATPs more tightly and would allow the generated proof to automatically be inserted next to the theorem in MMT. This would ensure the proof isn't recomputed and the type-checker would fail if the proof was no longer valid.

Plugin Integration At the moment the only way the IntelliJ Plugin UI can interact with out extension is via the "type-check" button, which triggers the ATP process when a `PROVE` is found. Better integration with the plugin could allow for triggering the proof search for a single declaration with `PROVE` instead of all of them at once. Also you could have an easy accessible toggle to disable the time and resource intensive proving process, when you're still editing parts of the theorem.

Change Detection Export to TPTP needs to be triggered manually, which then exports the targeted theories, regardless if they haven't changed since the last export. Detecting changes in the theories would reduce unnecessary exports, and more importantly could trigger the export automatically to be sure that in an attempt of automatic proving the included axiom files are up to date.

Improved Definition Handling The current handling of definitions is very lackluster and could be thoroughly rethought to allow a wider range of definitions.

Supporting more MMT Features Only a small amount of MMT features are compatible with the translation process in the moment. Supporting other existing and future features is of great value to the quality and applicability of the TPTP export process. Support for views could open completely new opportunities, as they could allow building views with TPTP-compatible logics interpreting a TPTP-foreign logic.

External ATP Cooperation LEO-III itself has the option[3] to pass on problems which it can't solve to other external ATPs. Due to LEO-III already using TPTP it can very easily interact with other provers taking TPTP input.

References

[GAA+13] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stèphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 163–179. Springer, 2013.

[3] `https://github.com/leoprover/Leo-III/blob/master/USAGE.md#enabling-external-cooperation`

[isa] Isabelle.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *SOSP*, 2009.

[Koh06] Michael Kohlhase. *OMDoc – An open markup format for mathematical documents [Version 1.2]*. Number 4180 in LNAI. Springer Verlag, August 2006.

[KU14] Cezary Kaliszyk and Josef Urban. Learning-assisted automated reasoning with Flyspeck. *J. Automat. Reason.*, 53(2):173–213, 2014.

[LAT] LATIN2 – logic atlas version 2.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

[MMT] OMDoc/MMT urtheories.

[MQP06] Jia Meng, Claire Quigley, and Lawrence C. Paulson. Automation for interactive proof: first prototype. *Inform. and Comput.*, 204(10):1575–1596, 2006.

[PB10] Lawrence C. Paulson and Jasmin Christian Blanchette. Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. In Geoff Sutcliffe, Eugenia Ternovska, and Stephan Schulz, editors, *International Workshop on the Implementation of Logics*, 2010.

[Rab18] Florian Rabe. Mmt: A foundation-independent logical framework. 2018.

[Tea03] The Coq Development Team. The coq proof assistant reference manual (version 7.4). Technical report, INRIA, Rocquencourt, France, 2003.

Acknowledgement

Thanks to Navid Roux for their time, patience and advice, spent on my questions and struggles.

Thanks to Alexander Steen for helping me out with the internals of LEO-III's tptp-parser and cooperating with me for the comment pull-request.