



Thousands of Problems to Lean

Laura Kovács¹  and Axel Polaczek¹ 

TU Wien, Vienna, Austria

Abstract

We provide a translation from problems of the Thousands of Problems for Theorem Provers (TPTP) problem library to `LEAN` expressions. This makes the TPTP library available as a benchmark set for proof automation tactics inside `LEAN`. We evaluate our approach and implementation by comparing our performance with that of the `LEAN` compiler on equivalent inputs. Furthermore, we use our framework to compare different `LEAN` proof automation tactics.

1 Introduction

The Thousands of Problems for Theorem Provers (TPTP) world [21] is the established infrastructure used by automated theorem provers (ATPs) for development and serves as an experimental playground [20]. The TPTP problem library provides instances for different logics, most notably (un)typed first-order logic, first-order arithmetic, and higher-order logic. Higher-order logic has been the reasoning home for interactive theorem provers (ITPs), acting as a meta-logic equipped with a rich type system. The genericity of such logical expressivity is evidenced by, for example, the increasing popularity of the `LEAN` interactive theorem prover [9], with emerging playgrounds for mathematics [22, 1], code reasoning [4], and cloud authorization [8]. Our work bridges ATP/ITP playgrounds: we provide access to the TPTP ecosystem within `LEAN` and turn the TPTP problem library into a benchmark for automation tools used within `LEAN`.

Converting ATP problems into ITP goals, and vice-versa, is an actively researched area. For instance, `LEAN-SMT` [13] makes use of the satisfiability modulo theories (SMT) solver `CVC5` [3] to solve goals in `LEAN`. On the other hand, `LEAN-AUTO` provides an encoding of `LEAN`'s dependent type theory into higher- and first-order logic, allowing this encoding to express goals as inputs for an ATP. `LEAN-AUTO` can use different ATPs as a backend. `DUPER` [6] is an ATP written in `LEAN` and provides one such backend for `LEAN-AUTO`. Another backend of `LEAN-AUTO` is the ATP `ZIPPERPOSITION` [7] for which `LEAN-AUTO` provides a parser of the Thousands of Solutions from Theorem Provers (TSTP) syntax output [19] by `ZIPPERPOSITION`, in order to perform proof replay in `LEAN`. One of the main challenges when using ATPs to solve goals in an ITP is premise selection, i.e., the choice of the known facts to include as axioms to the ATP. Tools which provide premise selection and make calls to ATPs are often referred to as hammers. In `ISABELLE/HOL` [15] a prominent hammer is given by `SLEDGEHAMMER` [16] which uses most notably the ATP `VAMPIRE` [5] as a backend. For proof reconstruction, `ISABELLE/HOL` uses the result of the ATP backend as a further relevancy filter to extract a list of axioms that are sufficient to prove the goal, and then uses the ATP `METIS` [10] designed specifically to integrate

with ISABELLE/HOL’s kernel. In LEAN a hammer is provided by LEANHAMMER [23] which makes use of AESOP [12], DUPER, GRIND and LEAN-AUTO.

LEAN’s standard library provides some automation in the form of tactics [11]. SIMP is a tactic that iteratively applies rewrite-rules until a fixed point is reached. GRIND is a tactic that performs SMT-inspired reasoning. In both cases the set of applicable rules can be extended or restricted by the user.

Our contributions. We extend the landscape of ATP/ITP interplay by connecting TPTP and LEAN, as follows. We develop a translation from TPTP instances to statements of theorems in LEAN (Section 3), by parsing (TPTPPARSER) and elaborating (TPTPTRANSULATOR) TPTP problems into LEAN theorems. Our translation takes place within LEAN, minimizing the dependency on external tools and making it easy for the LEAN community to utilize our translation (Section 2). We use the TPTP problem library as a benchmark for proof automation tools available in LEAN and compare them to the state-of-the-art in automated theorem proving (Section 4). Our experiments show that the automation tools available in LEAN prove thousands of problems, albeit just a fraction of what the state-of-the-art automated theorem provers can achieve.

2 Our Implementation from TPTP to LEAN

Within the context of an ATP, each TPTP problem can be viewed as a statement that can either be proven or refuted. Provable instances are those of main interest in the context of an ITP. To accommodate such instances in LEAN, we (i) develop TPTPPARSER and translate the abstract syntax tree of TPTP into a LEAN object of a type capturing the TPTP syntax. Further, we (ii) devise TPTPTRANSULATOR and use elaboration to turn LEAN expressions into LEAN theorems corresponding to TPTP theorems. We summarize below the implementation behind our approach.

Implementation. Our approach is implemented in the LEAN libraries TPTPPARSER and TPTPTRANSULATOR and is available at <https://doi.org/10.5281/zenodo.20679549>. The parser of TPTPPARSER is written in C, using the lexer FLEX and the parser generator BISON. To communicate with LEAN we make use of LEAN’s foreign function interface (FFI). The library TPTPTRANSULATOR has TPTPPARSER as a dependency.

(i) TPTPPARSER. The output of our parser is directly a LEAN object which is constructed by the C parser by making use of the reverse FFI. We further use the FFI to call the parser from inside LEAN. Our resulting tool is the LEAN library TPTPPARSER which provides a function to parse files in the TPTP format to their corresponding abstract syntax tree representation. Our parser handles all of the TPTP syntax constructs given in TPTP revision v9.2.1.0. One technical detail is that our implementation depends on the LEAN runtime and thus needs to be linked against the appropriate libraries. For the intended use case of using the library when running LEAN (interactively) via LAKE, linking is done automatically dynamically. However, if one wishes to produce a standalone binary one may link against TPTPPARSER statically by also linking against the LEAN runtime statically, which would result in rather large binaries. One possible alternative to using the FFI would be to make use of LEAN’s own parser which is in principle made to be highly extensible. In the case of the TPTP language this, however, would require the use of custom tokenizers¹, a decision which we chose not to take.

¹For instance, TPTP has special identifiers with a leading single quote. LEAN’s tokenizer interprets single

```
% Example of a TPTP problem, using the TPTP syntax
fof(a1, axiom, ! [X] : p(X)).
fof(b, conjecture, p(a)).
```

Figure 1: Illustrative TPTP example.

(ii) **TPTPTRANSLATOR**. The `LEAN` object constructed by our `TPTPPARSER` corresponds to the abstract syntax tree of the TPTP input file. This `LEAN` object is next translated to a `LEAN` expression by making use of the metaprogramming functions provided by the `LEAN` standard library. This step of our implementation is realized within our `LEAN` library `TPTPTRANSLATOR`: our library now provides functions to convert TPTP problem library instances from the first-order form (FOF), clause normal form (CNF), and typed first-order form (TFF) problem categories to `LEAN`.

3 Translating Thousands of Problems to `LEAN`

We discuss the main steps behind our TPTP/`LEAN` translation approach materialized in the `TPTPPARSER` and `TPTPTRANSLATOR` libraries of Section 2. For the ease of understanding, we illustrate our approach using the TPTP problem instance of Figure 1.

TPTP syntax to `LEAN` objects – `TPTPPARSER`. When `LEAN` processes a file, it first parses the file to an object of type `Lean.Syntax`. This syntax tree is generic and allows the labeling of nodes with arbitrary names. Similarly to this behavior, when we process a TPTP file, our `TPTPPARSER` converts the file into its abstract syntax tree as defined by the TPTP language specification. Internally in `TPTPPARSER`,

- we define for each nonterminal symbol of the (TPTP) grammar an inductive type, and
- the result of the translation is an object of the starting symbol `tptp_file`.

In contrast to `LEAN`’s internal representation of the `LEAN` syntax, our translation approach is to have the (TPTP) grammar reflected in the types. One advantage of this approach is that when operating on the resulting abstract syntax tree (AST), `LEAN`’s type system forces one to handle all cases. However, this leads to rather long compile times when compiling our `TPTPTRANSLATOR` library using the `LEAN` compiler, but this does not seem to pose a large overhead during runtime (see Section 4). One of the reasons of the long compile times is the presence of mutual inductive types with many member types, the largest of which consists of 32 types.

Example 1. For the example of Figure 1, `TPTPPARSER` produces an abstract abstract syntax tree whose partial structure is given below:

```
TptpFile.mk
  [TptpInput.annotatedFormula
    (AnnotatedFormula.fofAnnotated ...),
   TptpInput.annotatedFormula
    (AnnotatedFormula.fofAnnotated ...)]
```

quotes always as delimiters of `Char` literals.

Elaborating LEAN expressions – TPTPTRANSLATOR. When LEAN processes a file, the syntax is expressed in terms of `Lean.Syntax` objects. This syntax is transformed into LEAN expressions, i.e., objects of the type `Lean.Expr`, by a process referred to as *elaboration*. Similarly, our TPTPTRANSLATOR converts the abstract syntax tree obtained from TPTPPARSER to LEAN expressions, making use of the standard library functions provided by LEAN. We refer to this process within TPTPTRANSLATOR also as elaboration, as it also provides the transition from syntax to expression objects.

We convert the abstract syntax tree of a TPTP instance into a LEAN expression, as follows. Taking a simplified view of a TPTP problem library instance, a TPTP problem consists of any number of axioms $A_1, \dots, A_n$ and up to one conjecture B ; in case B is not explicitly stated, we consider it to be the truth value false ($\perp$). The semantics encoded in a TPTP problem file is essentially expressed by the question whether axioms A_i imply the conjecture B , i.e., whether

$$A_1, \dots, A_n \vdash B.$$

Let us denote by $\llbracket \cdot \rrbracket$ our translation function $\llbracket \cdot \rrbracket$ that maps TPTP formulas to LEAN expressions. Then, the type corresponding to the TPTP instance is the arrow type:

$$\llbracket A_1 \rrbracket \rightarrow \dots \rightarrow \llbracket A_n \rrbracket \rightarrow \llbracket B \rrbracket.$$

In the first step of the elaboration process of TPTPTRANSLATOR, we construct an expression for each annotated formula of the TPTP problem instance. During this process, a metavariable of the correct type is introduced for each symbol appearing in the TPTP instance. The result of this first translation step is a list of LEAN expressions containing metavariables.

Example 2. For the example of in Figure 1, TPTPTRANSLATOR yields the following state:

- *Axioms:* `[(a1 : ∀ (X : ?α), ?p X)]`
- *Conjecture:* `?p ?a`

The types of the metavariables are `?α : Type`, `?p : ?α → Prop` and `?a : ?α`.

The second step of the elaboration process of TPTPTRANSLATOR consists of creating an arrow type, beginning with the types and symbols appearing in the TPTP instance, and then followed by the axioms and the conjecture, where the metavariables are then assigned their bound expressions. This is achieved using the LEAN metaprogramming functions `Lean.Meta.withLocalDecls`, `Lean.Meta.mkForallFVars`, and their variants. The resulting LEAN proposition then corresponds to the (potential) theorem given by the TPTP instance.

Example 3. For the example of Figure 1, TPTPTRANSLATOR derives the following type

`∀ (α : Type) [Nonempty α] (p : α → Prop) (a : α), (a1 : ∀ (X : α), p X) → p a`

Example 3 highlights one technical detail of our translation: the typeclass constraint for the first-order type which we named α . Since first-order logic requires all models to have a nonempty universe, we must encode this also in our translation. Since TPTP is inherently classical on the instances we considered, we chose the LEAN typeclass `Nonempty` instead of `Inhabited`.

The TPTP instances we considered are based on classical logic. This entails the following technical detail for TFF problem instances of TPTP. Namely, TFF problems commonly use an interpreted ternary function symbol `$ite` corresponding to an if-then-else expression, where the first argument is a boolean condition and this, in TPTP, is expected to be an expression

evaluating to the interpreted type `$o`. We map this condition to `Prop`, as `$o` is also the type of predicates in TPTP. As we translate TPTP expressions `$ite` directly to LEAN `if ... then ... else ...` constructs, we make propositions appearing as the first argument of `$ite` decidable; we achieve this using `Classical.propDecidable` in LEAN.

For formulas in TFF, the TPTP standard [18] defines some interpreted symbols. Our TPTP-TRANSLATOR makes an effort to translate these interpreted symbols to semantically meaningful LEAN expressions. Most of the interpreted symbols are polymorphic over the interpreted types of integers, rationals and reals. In LEAN, polymorphism is expressible in terms of typeclasses and TPTPTRANSLATOR follows this LEAN approach. For operations such as the basic arithmetic operations of `$sum`, `$difference`, `$times` and `$quotient` we make use of the existing LEAN typeclasses `HAdd`, `HSub`, `HMul` and `HDiv`, respectively. For other function symbols for which no suitable typeclasses exist, we define our own typeclasses and provide suitable instances for the supported types. One symbol for which we were not able to carry out (yet) the translation meaningfully is `$to_rat` which is not (totally) defined on real numbers. Since LEAN does not unfold definitions automatically, we provide a tactic `tptp_translator_rw_defs` that unfolds all of our definitions, exposing the underlying definitions of the LEAN standard library and MATHLIB.

4 Evaluation

For the ease of readability and experimental evaluation of our approach, we denote by TPTP-LEAN our overall translation approach from TPTP to LEAN, and emphasize that TPTPLEAN arises from TPTPPARSER and TPTPTRANSLATOR. We evaluated our TPTPLEAN translation approach using a two-fold goal. Firstly, we evaluate the performance of our parser TPTPPARSER and of our elaborator TPTPTRANSLATOR by comparing it with the performance of LEAN. Secondly, we use our TPTPLEAN translation of TPTP problem instances in order to evaluate existing automation tactics available in LEAN: that is, we use the resulting LEAN theorems generated by our translation TPTPLEAN and evaluate existing LEAN tactics on these LEAN theorems.

Benchmark set. We considered 20 394 problems from the TPTP library, taking the instances from the categories of first-order form (FOF) (9259 instances), clause normal form (CNF) (8344 instances), typed monomorphic first-order form (TF0) (2113 instances) and typed polymorphic first-order form (TF1) (678 instances).

Experimental setup. All our experiments were carried out using an AMD Epyc 7502 2.5 GHz processor and 1 TB RAM. Experiments were run with a memory limit (as indicated later); we limited the number of concurrent instances to 64, which corresponds to the number of available cores in our experimental setting. The LEAN toolchain we used is `leanprover/lean4:v4.29.0`. LEAN was run with the command line options `-DmaxHeartbeats=0` and `--tstack=16000000`.

4.1 Performance

We compare the performance of our parser TPTPPARSER and of our elaborator TPTPTRANSLATOR with those of LEAN as follows. Given a TPTP benchmark instance, we run the LEAN compiler on a file containing a theorem whose type is the pretty-printed LEAN expression that our translation generated on the considered TPTP problem. This provides a comparison with an approach where the TPTP instance would be translated to LEAN syntax

by some other means. We use the LEAN compiler’s option `--profile` to obtain the time spent on parsing and elaboration. The results are shown in Figure 2, with the data being collected by running our translation approach and LEAN with a 60 s timeout. We note that not all instances that could be elaborated by our translation method within the timeout are shown, since in the timeout we also included the time it takes for pretty-printing the generated expression which can take a significant amount of time for large instances. The times shown in the graph are only parsing and elaboration times and do not include pretty-printing.

By reflecting on the parsing results of Figure 2, the LEAN parser is more efficient on small instances than our TPTPPARSER; yet, for large instances, our TPTPPARSER performs better. For the speedup of parsing with TPTPPARSER when compared to LEAN we observed quartiles $Q_1 = 3.8$, $Q_2 = 8.6$ and $Q_3 = 21.0$. The TPTP instance with the largest speedup for parsing was [SEU406+2](#) with a speedup of 3073.8. On 4.8 % of instances LEAN parsed the input faster than TPTPPARSER, where the greatest speedup observed was for the TPTP instance [ANA013-2](#) with a speedup of 11.0.

For elaboration, the difference between our TPTPTRANSLATOR elaborator and LEAN is not quite as pronounced, but it appears that for most instances our TPTPTRANSLATOR elaborator is faster. For some instances, our approach elaborates the expressions much faster than the LEAN elaborator. For the speedup of elaboration with TPTPTRANSLATOR when compared to LEAN we observed quartiles $Q_1 = 1.4$, $Q_2 = 2.0$ and $Q_3 = 2.5$. The instance with the largest speedup for elaboration was [ITP353_1](#) with a speedup of 104.6. On 7.0 % of instances LEAN elaborated the expressions faster than TPTPTRANSLATOR, where the greatest speedup observed was for instance [LCL223-3](#) with a speedup of 3.4.

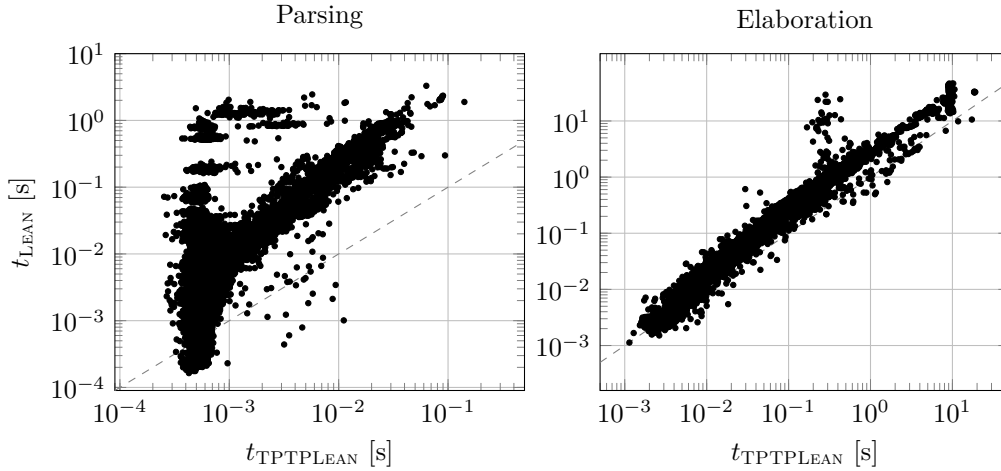


Figure 2: A comparison of the time spent generating LEAN expressions corresponding to TPTP problem instances, using our TPTPLEAN translation approach based on our parser TPTPPARSER and elaborator TPTPTRANSLATOR. The x -axis corresponds to the overall time spent by our parser and elaborator to parse a TPTP file and construct the corresponding LEAN expression. The y -axis corresponds to the time spent by LEAN to parse and elaborate a LEAN file corresponding to the pretty-printed output of the LEAN expression our translation constructed.

4.2 Experimental Comparisons and Results

Given our benchmark set, we were interested to evaluate state-of-the-art theorem provers that natively parse TPTP against the various automation tactics for first-order reasoning available in LEAN. In particular, we compared the LEAN tactics supporting first-order reasoning against the ATP VAMPIRE [5]. Given a TPTP problem instance, we thus ran VAMPIRE on the TPTP problem, as well as ran LEAN tactics on the LEAN theorem generated by our TPTPLEAN translation. The VAMPIRE version we used has a GIT commit hash `9eeba91d14fa99f7fe984b1e27ae26e4c323bc3f`. VAMPIRE was run with the command line options `--proof off` and `--mode portfolio`. Our experiments shown in this section were performed with a 60s timeout and a memory limit of 16 GB. For LEAN tools startup and import times are excluded from the times reported below, but are included in the timeout. The LEAN tools we included in this analysis are GRIND, DUPER, LEAN-AUTO, LEAN-SMT and SIMP (see Section 1).

For the LEAN proofs we essentially used LEAN to check files of the form

```
example : load_tptp "/path/to/tptp/instance" := by tac
```

where the tactic `tac` is simply `simp` for SIMP and for the other LEAN-based tools is the tactic sequence

```
try tptp_translator_rw_defs
try intros
try tac'
```

where `tac'` is `grind` for GRIND, `duper [*]` for DUPER [6], `smt [*]` for LEAN-SMT [13] and `auto [*]` for LEAN-AUTO [17]. Note that the tactic `tptp_translator_rw_defs` is used to unfold all our definitions that are used for interpreted function symbols of TPTP. In the case of SIMP this is not needed since we marked all definitions with the `simp` attribute.

Table 1 shows the total number of TPTP instances solved by the different tools we considered. The instances shown for VAMPIRE are only those where VAMPIRE reported a refutation, i.e., does not include those instances which VAMPIRE proved to be satisfiable, removing a total of 1454 instances on which VAMPIRE terminated under the given resource constraints.

Tool	FOF	CNF	TF0	TF1
VAMPIRE	6408	5322	1370	309
GRIND	1754	1029	508	162
DUPER	1881	1708	253	86
LEAN-SMT	3010	2493	461	0
LEAN-AUTO	1546	1624	182	28
SIMP	91	7	176	3

Table 1: Total number of TPTP problem library instances solved within the time limit of 60s and a memory limit of 16 GB. For all tools but VAMPIRE, the TPTP problems have been translated to LEAN theorems using our translation.

Figure 3 shows the number of instances solved as a function of time. For the LEAN-based automation tactics, the time shown is measured starting with reading the input file and ending once the chosen tactic has terminated. For VAMPIRE we used portfolio mode and disabled proof output. One striking feature is that SIMP is able to solve many instances in a short amount of time. Some of those instances were propositional tautologies and others were instances that had an equality as the main goal which holds definitionally which SIMP is able to solve very quickly.

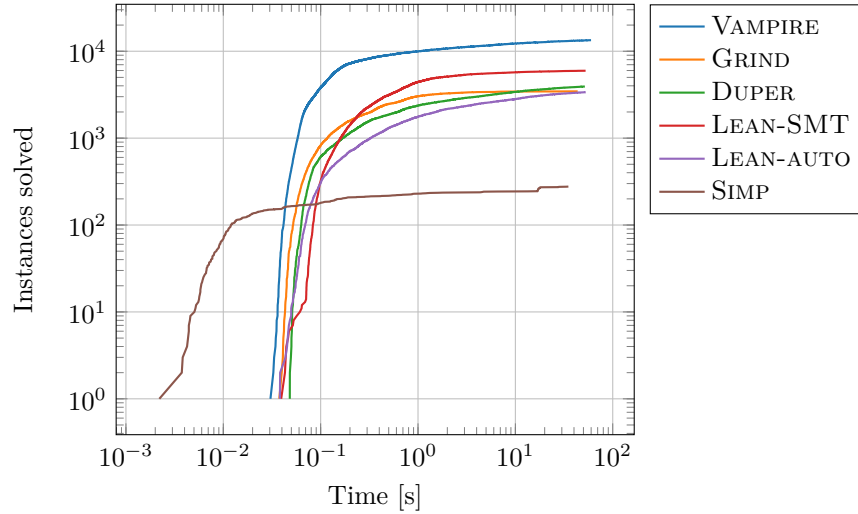


Figure 3: Cumulative number of TPTP instances solved as a function of time for the different tools as a function of time.

In fact, for the cases where the theorem holds purely by definitional equality, also our custom tactic `tptp_translator_rw_defs` solves this goal immediately. However, the implementation of that tactic is done via `simp only [...]`, where the ellipsis is a list of all our defined symbols for interpreted function symbols, which incurs a large overhead compared to a direct call of `simp`, where the definitions have been marked with the `simp` attribute.

While GRIND, DUPER, LEAN-SMT and LEAN-AUTO solve a comparable number of instances in the available time, they have different strengths, as shown in Table 2. When compared to VAMPIRE, we note the superiority of equality reasoning based on resolution and superposition [2, 14]: for example, VAMPIRE solved 9989 problems that GRIND could not solve, whereas GRIND solved 33 problems that VAMPIRE failed proving.

Interestingly, Table 2 shows that SIMP solves some instances which other LEAN tools were not able to solve. One issue is the usage of `intros` above which converts goals of the form $\vdash h_1 \rightarrow \dots \rightarrow h_n \rightarrow c$ into goals of the form $h_1, \dots, h_n \vdash c$. For some instances, after applying the tactic `intros`, the tactic `simp` is no longer to solve the goal. However, we also observed instances for which the usage of `intros` before applying one of the tools' custom tactic was crucial. Since having the hypotheses separated from the conjecture is closer to the spirit of TPTP we decided to use `intros` for all tools except SIMP. Section 4.3 further details the effect of using `intros`.

4.3 Experimental Analysis of Using the `intros` Tactic

We finally investigate the impact of using the `intros` tactic for the LEAN tactics we consider. The tactic sequence we employ when using `intros` is

```
try intros
try tac
```

where `tac` is `grind` for GRIND, `duper [*]` for DUPER, `smt [*]` for LEAN-SMT, `auto [*]` for LEAN-AUTO, and `simp [*] at *` for SIMP. In the case of GRIND, DUPER, and LEAN-SMT, LEAN-AUTO, GRIND the tactic sequence is prepended with our unfolding tactic `try`

	VAMPIRE	GRIND	DUPER	LEAN-SMT	LEAN-AUTO	SIMP
VAMPIRE	–	9989	9482	7533	10030	13133
GRIND	33	–	1107	662	1371	3205
DUPER	1	1582	–	782	706	3712
LEAN-SMT	88	3173	2818	–	3156	5763
LEAN-AUTO	1	1298	158	572	–	3216
SIMP	1	29	61	76	113	–

Table 2: Comparison of tool advantages regarding TPTP instances solved within a 60 s timeout. An entry n in row R and column C indicates that tool R solved n instances that tool C was unable to solve.

`tptp_translator_rw_defs`. The case without `intros` is the same as above with the tactic `try intros` removed and in the case of SIMP the tactic `simp [*] at *` is replaced by `simp`. Note that the results shown in the main text use `intros` for GRIND, DUPER, and LEAN-SMT, LEAN-AUTO, and GRIND, whilst not using `intros` for SIMP. The reason for not using `intros` for SIMP is that we wanted to contrast the results with a simple automating tactic such as `simp`.

In Table 3 we show the number of TPTP problem library instances solved by the different tools for both the tactic sequence with `intros` and without `intros`. In all cases the number of instances solved with `intros` is greater, which justifies the choice of this sequence in the main text. However, the table also shows that by using `intros` some instances are no longer solved within the resource constraints given. This effect is most pronounced for DUPER, LEAN-SMT and SIMP.

Tool	w/o	w/	$w/o \wedge \overline{w/}$	$w/ \wedge \overline{w/o}$
GRIND	3372	3452	18	98
DUPER	3850	3930	125	205
LEAN-AUTO	3380	3381	1	2
LEAN-SMT	5938	5975	151	188
SIMP	277	565	77	365

Table 3: Comparison of the number of TPTP problem library instances solved with or without the `intros` tactic by different LEAN tools within a 60 s time limit and a 16 GB memory limit. The second column shows the number of instances solved without using the `intros` tactic and the third column shows the number of instances solved with using the `intros` tactic. The fourth column show the number of instances that were solved with using the `intros` tactic but not without using the `intros` tactic and the fifth column show the number of instances that were solved without using the `intros` tactic but not with using the `intros` tactic.

Figure 4 shows a comparison of the runtimes for the different LEAN tools with and without using the `intros` tactic.

5 Conclusions

Our work is a step in furthering the interaction of ATPs and LEAN. By making the TPTP problem library available in LEAN it can serve as a benchmark for automation in LEAN. We compared different tools for automation in LEAN to showcase their strengths on TPTP problem

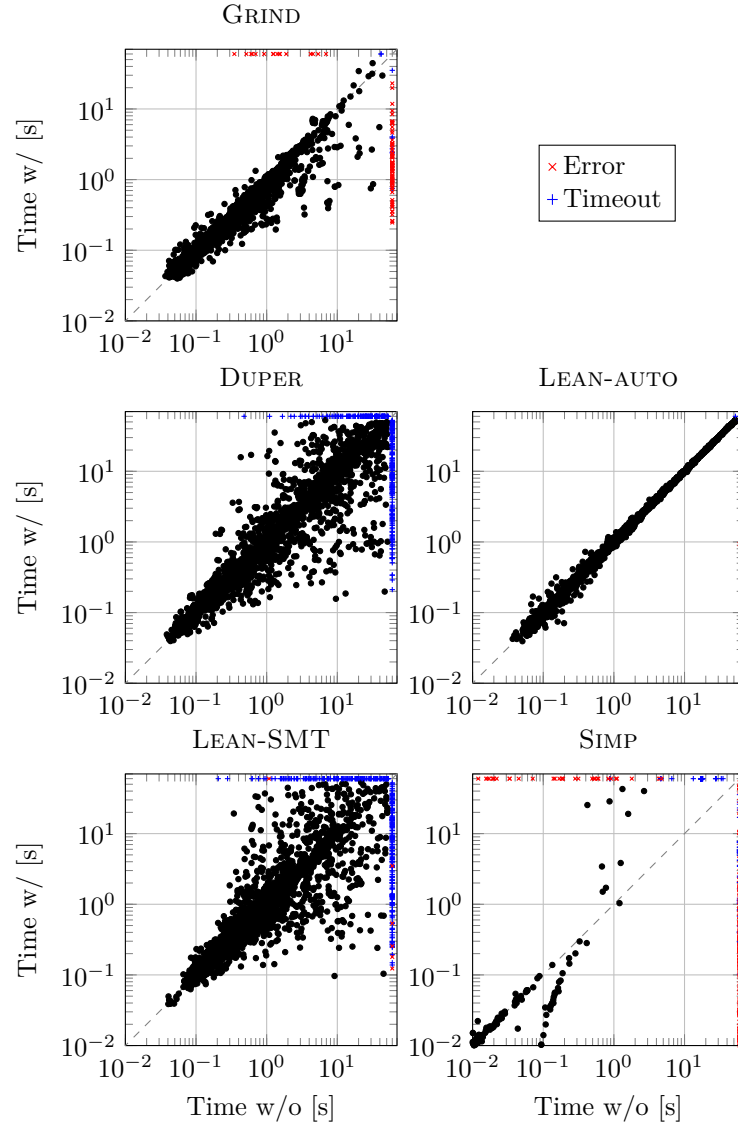


Figure 4: Comparison of the runtime for solving the instances with or without using the `intros` tactic. Per instance there was a 60s time and a 16 GB memory limit.

instances. However, comparing the results to those of the ATP VAMPIRE indicates that there is room for improvement, possibly by leveraging external tools. Future work includes extending our translation approach to higher-order and dependently typed logic, making all of the TPTP problem library available in LEAN. One potential use case of our translation approach is to facilitate proof reconstruction when using an ATP as a hammer that outputs its proof in the TSTP format.

Acknowledgements. This work was partially funded by the ERC Proof of Concept Grant LEARN 101213411, the Austrian Science Fund – FWF [Grant ID 10.55776/EFP2661725], and the SBA Research COMET Center SBA-K1 NGC managed by the FFG.

References

- [1] Anne Baanen, Matthew Robert Ballard, Johan Commelin, Bryan Gin-ge Chen, Michael Rothgang, and Damiano Testa. Growing Mathlib: Maintenance of a Large Scale Mathematical Library. In *CICM*, pages 51–70. Springer, 2026. doi:10.1007/978-3-032-07021-0_4.
- [2] Leo Bachmair and Harald Ganzinger. Resolution Theorem Proving. In *Handbook of Automated Reasoning*. North-Holland, 2001. doi:10.1016/B978-044450813-3/50004-7.
- [3] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *TACAS*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022. doi:10.1007/978-3-030-99524-9_24.
- [4] Clark Barrett, Swarat Chaudhuri, Fabrizio Montesi, Jim Grundy, Pushmeet Kohli, Leonardo de Moura, Alexandre Rademaker, and Sorrachai Yingchareonthawornchai. CSLib: The Lean Computer Science Library, 2026. arXiv:2602.04846.
- [5] Filip Bártek, Ahmed Bhayat, Robin Coutelier, Márton Hajdú, Matthias Hetzenberger, Petra Hozová, Laura Kovács, Jakob Rath, Michael Rawson, Giles Reger, Martin Suda, Johannes Schoisswohl, and Andrei Voronkov. The Vampire Diary. In *CAV*, volume 15933 of *Lecture Notes in Computer Science*, pages 57–71. Springer, 2025. doi:10.1007/978-3-031-98682-6_4.
- [6] Joshua Clune, Yicheng Qian, Alexander Bentkamp, and Jeremy Avigad. Duper: A Proof-Producing Superposition Theorem Prover for Dependent Type Theory. In *ITP*, volume 309 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.ITP.2024.10.
- [7] Simon Cruanes. *Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond. (Extensions de la Superposition pour l’Arithmétique Linéaire Entière, l’Induction Structurale, et bien plus encore)*. PhD thesis, École Polytechnique, Palaiseau, France, 2015. URL: <https://tel.archives-ouvertes.fr/tel-01223502>.
- [8] Joseph W. Cutler, Craig Disselkoen, Aaron Eline, Shaobo He, Kyle Headley, Michael Hicks, Kesha Hietala, Eleftherios Ioannidis, John Kastner, Anwar Mamat, Darin McAdams, Matt McCutchen, Neha Rungta, Emina Torlak, and Andrew M. Wells. Cedar: A New Language for Expressive, Fast, Safe, and Analyzable Authorization. *Proc. ACM Program. Lang.*, 8(OOPSLA1):670 – 697, 2024. doi:10.1145/3649835.
- [9] Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In *CADE*, volume 12699 of *Lecture Notes in Computer Science*, page 625–635. Springer-Verlag, 2021. doi:10.1007/978-3-030-79876-5_37.
- [10] Joe Hurd. First-Order Proof Tactics in Higher-Order Logic Theorem Provers. In Myla Archer, Ben Di Vito, and César Muñoz, editors, *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003)*, number NASA/CP-2003-212448 in NASA Technical Reports, pages 56–68, 2003. URL: <http://www.gilith.com/papers>.
- [11] Lean FRO. The Lean Language Reference – Tactic Reference. <https://lean-lang.org/doc/reference/4.29.0/Tactic-Proofs/Tactic-Reference/>. Accessed: 2026-06-17.
- [12] Jannis Limperg and Asta Halkjær From. Aesop: White-Box Best-First Proof Search for Lean. In *CPP*, page 253–266. ACM, 2023. doi:10.1145/3573105.3575671.
- [13] Abdalrhman Mohamed, Tomaz Mascarenhas, Harun Khan, Haniel Barbosa, Andrew Reynolds, Yicheng Qian, Cesare Tinelli, and Clark W. Barrett. lean-smt: An SMT Tactic for Discharging

- Proof Goals in Lean. In *CAV*, volume 15933 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 2025. doi:10.1007/978-3-031-98682-6_11.
- [14] Robert Nieuwenhuis, Thomas Hillenbrand, Alexandre Riazanov, and Andrei Voronkov. On the Evaluation of Indexing Techniques for Theorem Proving. In *IJCAR*, 2001. doi:10.1007/3-540-45744-5_19.
 - [15] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. doi:10.1007/3-540-45949-9.
 - [16] Lawrence C. Paulson and Jasmin Christian Blanchette. Three Years of Experience with Sledgehammer, a Practical Link Between Automatic and Interactive Theorem Provers. In *IWIL*, volume 2 of *EPiC Series in Computing*, pages 1–11. EasyChair, 2012. doi:10.29007/tnfd.
 - [17] Yicheng Qian, Joshua Clune, Clark W. Barrett, and Jeremy Avigad. Lean-Auto: An Interface Between Lean 4 and Automated Theorem Provers. In *CAV*, volume 15933 of *Lecture Notes in Computer Science*, pages 175–196. Springer, 2025. doi:10.1007/978-3-031-98682-6_10.
 - [18] Geoff Sutcliffe. TPTP technical report. <https://tptp.org/UserDocs/ProblemLibraryManual/TPTPTR.shtml>. Accessed: 2026-06-13.
 - [19] Geoff Sutcliffe. The TPTP World – Infrastructure for Automated Reasoning. In *LPAR*, number 6355 in *Lecture Notes in Artificial Intelligence*, pages 1–12. Springer-Verlag, 2010. doi:10.1007/978-3-642-17511-4_1.
 - [20] Geoff Sutcliffe. The CADE ATP System Competition — CASC. *AI Magazine*, 37(2):99–101, 2016. doi:10.1609/aimag.v37i2.2620.
 - [21] Geoff Sutcliffe. The Logic Languages of the TPTP World. *Logic Journal of the IGPL*, 2022. doi:10.1093/jigpal/jzac068.
 - [22] The mathlib Community. The Lean Mathematical Library. In *CPP*, page 367–381. ACM, 2020. doi:10.1145/3372885.3373824.
 - [23] Thomas Zhu, Joshua Clune, Jeremy Avigad, Albert Qiaochu Jiang, and Sean Welleck. Premise Selection for a Lean Hammer, 2026. arXiv:2506.07477.