Proceedings of the first Proof Verification Competition (ProoVer 2026)

Julie Cailler[1] and Simon Guilloud[2]

[1] University of Lorraine, CNRS, Inria, LORIA, Nancy, France
[2] EPFL, Lausanne, Switzerland

Abstract

The ProoVer competition is an annual evaluation of automatic proof checkers — systems that verify whether a candidate proof is correct w.r.t. its conjecture. ProoVer 2026 was the first edition of the competition. Ten proof checkers competed. These proceedings present the competition design and information about the competing systems.

1 Introduction

The Proof Verification Competition (ProoVer 2026) is the first edition of the annual competition for proof checkers. Proof checkers are systems that verify whether a candidate proof is correct, meaning that it derives conjecture from a set of axioms through intermediate step that each correctly follow from their premises. One purpose of ProoVer is to provide a public evaluation of the current capabilities of proof checking systems, in terms of both correctness and robustness. Additionally, ProoVer aims to stimulate research and the development of robust, reusable proof checkers, to encourage interactions between people working on proof checking and automated deduction, and to give the community an occasion to discuss standards for proofs — what should count as a valid proof step, and how proofs should be represented, justified, and communicated between systems.

In ProoVer, participants are submitted both correct and "buggy" proofs, and their systems must determine the validity of each proof. Contestants are required to implement a proof checker that can call an external prover for some steps, while verifying specific proof steps internally within their tool. ProoVer evaluates the performance of proof checkers in terms of how accurately, and how quickly, they classify proofs as valid or invalid, in the context of a fixed set of proofs and a per-proof time limit. The grading scheme is detailed in Section 5.

ProoVer is a new competition, inspired by and organized alongside CASC, the CADE ATP System Competition [23]. The two competitions share their infrastructure, computers, and much of their organizational machinery (see Section 3), and ran in parallel at ProoVer 2026, but are judged and ranked independently. ProoVer's proof are expressed in the TSTP proof format [26].

ProoVer is affiliated with IJCAR (the International Joint Conference on Automated Reasoning) conference. ProoVer 2026 was held on 27th July 2026, as part of the 13th International Joint Conference on Automated Reasoning (IJCAR 2026), which in turn was part of the Federated Logic Conference 2026 (FLoC 2026), in Lisbon, Portugal.

ProoVer 2026 was organized by Julie Cailler and Simon Guilloud, assisted by Geoff Sutcliffe for the technical parts, and overseen by a panel consisting of Cláudia Nalon, Aart Middeldorp, and Martina Seidl. The competition was run on computers provided by the StarExec project [19] at the University of Miami.

The ProoVer 2026 web site `https://proover-competition.github.io/` provides access to all the resources used before, during, and after the event.

The ProoVer rules, specifications, and deadlines are absolute. Only the panel has the right to make exceptions. It is assumed that all entrants have read the documentation related to the competition, and have complied with the competition rules. Non-compliance with the rules can lead to disqualification. A "catch-all" rule is used to deal with any unforeseen circumstances: *No cheating is allowed*. The panel is allowed to disqualify entrants due to unfairness, and to adjust the competition rules in case of misuse.

These proceedings are organized as follows: Section 2 lists the divisions of ProoVer 2026, together with proof checkers involved in ProoVer 2026 and their entrants. Section 3 describes the competition infrastructure. Section 4 describes the proof format and the specified proof steps. Section 5 describes how the systems are evaluated. Section 6 describes the practical steps for entering a system. Section 7 lists system properties that entrants should check. Section 8 provides descriptions (written by the entrants) of the systems. Section 9 concludes.

2 Entrants & Divisions

ProoVer 2026 entrants compete in one of two divisions: the regular division, in which systems are ranked according to the grading scheme of Section 5, and the demonstration division, in which systems are run and reported on but not ranked. Ten systems competed in the two divisions of ProoVer 2026.

2.1 The Regular Division

In this first edition, the regular division considers only problems and proofs expressed in (untyped) first-order form (FOF): every problem in the ProoVer 2026 problem set is an untyped first-order problem, and every proof a checker is asked to validate is an untyped first-order TSTP derivation. Extending the competition to other logics, e.g., typed first-order or higher-order form, is left to future editions.

Every entrant is a proof checker, and every proof checker is evaluated on the same set of problems and proofs. Eight proof checkers were entered into the regular division of ProoVer 2026. The systems and their entrants are listed in Table 1. System descriptions, written by the entrants, are in these proceedings (Section 8) and on the TPTP web site.

Table 1: The regular division proof checkers and entrants

Proof Checker	Entrant	Entrant's Affiliation
CheckProof 0.1	Nik Murzin	Wolfram Institute, USA
GAPT 2.20	Fabian Achammer & Martin Riener	TU Wien, Austria
mrs-proover 0.2.0	Olivier Roland	Independent Researcher, France
Nörgler 1.1	Alexander Steen, Melanie Taprogge & Happy Sariyanto	University of Greifswald, Germany
ProofCheck 1.0	Jeff Machado & Larry Lesyna	Independent Researcher, USA LXL Technology
ProofGuard 1.0	Matthew Farah & Alicia Marinache	McMaster University, Canada
PyCheck 0.1	Stephan Schulz	DHBW Stuttgart, Germany
VaLeaDate 0.1	Jonas Bodingbauer & Laura Kovacs	TU Wien, Austria

2.2 The Demonstration Division

Proof checkers that cannot be entered into the regular division for any reason, e.g., the entrant is an organizer of the competition, or the system requires special hardware or internet connection, can be entered into the demonstration division instead. The demonstration division uses the same problem and proof set as the regular division. Demonstration division systems can run on the competition computers, or on computers supplied by the entrant. The demonstration division's results are presented along with the regular division's results, but might not be directly comparable with them, and demonstration division systems are not ranked and are not eligible for the ProoVer 2026 trophy.

Two proof checkers were entered into the regular division of ProoVer 2026. The systems and their entrants are listed in Table 2. System descriptions, written by the entrants, are in these proceedings (Section 8) and on the ProoVer 2026 web site.

Table 2: The demonstration division proof checkers and entrants

Proof Checker	Entrant	Entrant's Affiliation
GDV 2.0	Geoff Sutcliffe	University of Miami, USA
GDV-LP 2.0	Frédéric Blanqui & Geoff Sutcliffe	ENS Paris-Saclay, INRIA, France University of Miami, USA

3 Infrastructure

3.1 Computers

The StarExec Miami computers used for the competition have:

- Two octa-core Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz CPUs, with hyperthreading (two threads per core).
- 128GiB memory.
- The CentOS Linux release 7.4.1708 (Core) operating system, Linux kernel 3.10.0-693.el7.x86_64.

One proof checker runs on one CPU at a time. StarExec uses Linux's `sched_setaffinity` to restrict each system run to a single CPU, and `setrlimit` to limit memory use to 128 GiB. This separation avoids contention that might affect performance measurements. Systems can use all the cores on the CPU. No CPU time limit is imposed, so that it can be advantageous for a proof checker to use all the cores on its CPU; instead, a wall clock time limit is imposed for each proof (see Section 3.2).

The StarExec Miami computers used for ProoVer are the same as are publicly available to the TPTP community, which allows system developers to test and tune their systems in exactly the same environment as is used for the competition.

3.2 Time Limits

Each proof checker is allowed a wall clock time limit of 30 seconds to reach a verdict on each proof, whether it is entered into the regular division or the demonstration division (Section 2). Proofs are evaluated independently of one another: a checker's performance on one proof has no bearing on the time or resources available to it on any other proof.

3.3 Problems and Proofs

Each benchmark given to a proof checker consists of one problem file, containing a set of axioms and a conjecture, and a corresponding proof file. Problem files are First-Order Logic (FOF) TPTP files, and are located in a `Problems/` directory. Proofs are written in the TSTP format [26]; see the TSTP Quick Guide at `tptp.org/UserDocs/QuickGuide/Derivations.html`.

A proof file's header contains a `Proof:` field giving the path to its corresponding problem file. Axioms and the conjecture are imported into a proof from its problem file via a `file` directive, e.g. `fof(a1, axiom, ![X]: p(X), file('Problems/example_c.p', a1))`. In every problem, whether accompanied by a valid or an invalid proof, the file named in such a directive is guaranteed to match the file named in the proof's `Proof:` header field, so a checker can rely on this without needing to parse the header field itself. Finally, a file `results.csv` containing the status of each proof is provided separately in order to check the answer of the proof checker.

4 Proof Format

4.1 Proof Steps

A proof is a sequence of inference steps. All proofs are proofs by refutation: the conjecture is negated, and the proof is required to end with `$false`. Each proof step has a role (`axiom`, `conjecture`, `negated_conjecture`, or `plain`) and a status (`thm`, `esa`, or `cth`). Proofs are expected to have reasonable granularity, i.e., to be checkable by a standard ATP system.

Some inference rules (detailed in Section 4.2) must be verified internally by the proof checker itself. All other (unspecified) inference steps may instead be validated by invoking an external ATP system, but only when the step's status is `thm` or `cth`. When doing so, the checker may give the external ATP only that step and its stated premises, and it is forbidden to re-check the whole (negated) conjecture independently, as a separate sanity check, using an external ATP.

There is no requirement on the order in which proof steps appear, and every problem and proof file provided to entrants is guaranteed to be syntactically well-formed and parsable as TPTP/TSTP.

4.2 Specified Rules

Not all problematic cases that a proof checker may encounter are explicitly covered by these rules. Entrants are responsible for handling edge cases, and for deciding what should, or should not, be considered an acceptable proof, in the spirit of the TSTP standard and of general proof-checking principles.

Skolemization

A skolemization step must:

- use the inference rule name `skolemize`;
- have status `esa`;
- introduce exactly one new Skolem symbol, declared with `new_symbols(skolem, [sK])`;
- explicitly indicate, with `skolemize(Var, sK(...))`, which existentially quantified variable `Var` is being eliminated, and the Skolem term replacing it.

The Skolem term must depend on at least the universally quantified variables that appear within the skolemized existential formula. The resulting formula must be a correct skolemization of its parent formula.

Negated Conjecture

A negated-conjecture step must:

- use the inference rule name `negated_conjecture`;
- have status `cth`;

- have a parent that is a `conjecture` step;
- have a formula that is apha-equivalent to the negation of its parent's formula – this may be checked either internally, or using an external ATP.

Axioms and Conjecture
An axiom or conjecture step must:

- have role `axiom` or `conjecture`;
- import its formula from the corresponding problem file via a `file` directive (see Section 3.3);
- have a formula that is alpha-equivalent to the formula named in the `file` directive within the referenced problem file – this may be checked either internally, or using an external ATP.

The name given to a step in the proof file may differ from the name of the corresponding axiom or conjecture in the problem file; the checker must verify that the referenced item exists in the problem file and that the two formulae are alpha-equivalent.

5 System Evaluation

For each proof checker, for each problem, the following items of data are recorded:

- whether or not the proof was verified, and the corresponding result
- the CPU and wall clock times prepended taken to each line of the system's `stdout` by StarExec's `runsolver` utility [11].

The systems are ranked according to the following grading scheme:

- Identifying a bad proof as bad = +2
- Identifying a good proof as good = +1
- Giving up = 0
- Identifying a good proof as bad = -1
- Identifying a bad proof as good = -10

The final score is the sum over all problems. Ties are broken according to the average time taken over problems solved. In the demonstration division, the systems are not ranked.

In addition to the ranking data, other measures are made and presented in the results:

- The *number of unique* quantifies the unique abilities of each system. For each problem solved by a system, its number of unique indicates the proofs it verifies but that are not verified by all other systems.
- The *trustworthiness* indicate the number of bad proofs being told good (false positive).

6 System Entry, Delivery, and Execution

To be entered into ProoVer 2026, a proof checker must be registered, in either the regular division or the demonstration division (Section 2), using the ProoVer 2026 system registration form by the system registration deadline. For each system entered, an entrant must be nominated to handle all issues (e.g., installation and execution difficulties) arising before, during, and after the competition, and that entrant must formally register for ProoVer 2026. It is not necessary for entrants to physically attend the competition, although it is more fun to do so. Entrants who are also entering CASC only need to pay the registration fee once for both competitions.

The proof checker itself, together with the command line needed to launch it, must be delivered to the competition organizers by the system delivery deadline. The installation must contain only the components necessary for running the checker; source code, if not already public, may be delivered separately. Entrants are encouraged to make a public release of their proof checker after the competition,

under an open source license, so that the checker can be freely used, modified, and shared, and so that its results remain reproducible.

Execution of the proof checkers is controlled by the competition organizers on the competition computers (Section 3.1). The competition computers copy the systems and problems to the compute nodes before starting execution, and timing starts when execution starts. Any command line parameters have to be the same for every problem. Proof checkers entered into the demonstration division are instead executed and supervised by their entrants (Section 2).

After the competition, all proofs are made available on the ProoVer 2026 website, and proof checkers are made available on the CASC web site, together with their source code where the entrant has agreed to release it, so that anyone can examine and reuse them. In the demonstration division, the entrant specifies whether or not the source code is placed on the web site.

6.1 System Descriptions

A system description must be provided for each proof checker, using the schema published on the ProoVer 2026 web site. The schema has four parts:

- *Overview* – introduces the proof checker and its main design choices, including its checking strategy (e.g., when it reports `Unknown` versus attempting additional verification steps), and any other notable features.
- *Implementation* – describes the implementation: the programming language used, important internal data structures, and any special libraries; it specifies any backend ATP or other external tools used, the availability of the system, and gives a brief explanation of how proofs are verified.
- *Expected Competition Performance* – makes predictions about the checker's likely performance.
- *References*.

The system description must be emailed to the competition organizers by the system description deadline, and forms part of these proceedings (Section 8).

6.2 Sample Solutions

Representative sample solutions – i.e., the SZS output that a checker produces when run on the published example problems (`COR001+1.s`, `EVL001+1.s`, and `TMO001+1.s`) – must be emailed to the competition organizers by the sample solutions deadline. Samples must be produced by running the checker on the example problems published on the ProoVer 2026 web site. An explanation must be provided for unexpected answer.

7 System Properties

Entrants must ensure that their proof checkers execute correctly in the competition environment, and have the following properties. Entrants are advised to finalize their installation packages and check these properties well in advance of the system delivery deadline, so that the organizers have time to help resolve any difficulties encountered.

Execution

- Proof checkers must be fully automatic.
- Proof checkers' performances must be reproducible by running the system again. .
- Proof checkers do not have to be correct in any sense: the main purpose of the competition being to stress edge cases, it is expected for proof checkers to have omitted some details during the implementation.
- Proof checkers do not have to be complete in any sense: reporting `Unknown` on a proof step, or a whole proof, that the checker cannot decide within the time limit is always an acceptable outcome, and is preferable to an incorrect verdict.

- All techniques used must be general purpose, and expected to extend usefully to new, unseen proofs. The precomputation and storage of information about individual competition problems, proofs, or their solutions, is not allowed. Strategies and strategy selection based on individual problems or their proofs are not allowed. If machine learning procedures are used to tune a system, the learning must ensure sufficient generalization that there is no specialization to individual problems; the system description must explain any such tuning or training. The competition panel may disqualify any system deemed to be problem specific rather than general purpose.

Output

- All solution output must be to `stdout`.
- For each input problem, the proof checker must output exactly one of the following distinguished SZS status lines [22]:

  ```
  % SZS status VerifiedGood
  % SZS status VerifiedBad
  % SZS status Unknown
  % SZS status Timeout
  ```

  `VerifiedGood` and `VerifiedBad` report that the checker has determined the proof to be, respectively, valid and invalid; `Unknown` reports that the checker could not reach a verdict; `Timeout` reports that the checker ran out of time. Any other output is treated as a failure on that proof.
- Additional information may be appended to the status line after a colon, e.g.

  ```
  % SZS status VerifiedBad : Step s2 is incorrect
  ```

- Solutions may not have irrelevant output (e.g., from other threads running in parallel) interleaved with the SZS status line.

Resource Usage

- Proof checkers running on the competition computers must be interruptible by a `SIGALRM` signal, so that the wall clock time limit (Section 3.2) can be imposed. For checkers that create multiple processes, the signal is sent first to the process at the top of the process hierarchy, then one second later to all processes, even if they have disconnected from the process hierarchy. The default action on receiving this signal is to exit, thus complying with the time limit; checkers may instead catch the signal and exit of their own accord. A checker that runs past the time limit is considered not to have reached a verdict on that problem, and is scored as `Timeout`.
- If a checker terminates of its own accord, it may not leave files anywhere. If a checker is terminated by `SIGALRM`, it may not leave files anywhere other than in `/tmp`.
- For practical reasons, excessive output from a proof checker is not allowed. A limit, dependent on the disk space available, is imposed on the amount of output that can be produced.

8 The Verifiers

These system descriptions were written by the entrants.

8.1 CheckProof 0.1

Nik Murzin
Wolfram Institute, USA

Overview

CheckProof 0.1 [10] is a checker for first-order TSTP proofs. A proof is validated by replay: it is parsed into its sequence of annotated formulae, and each derived step is validated independently by dispatching on the inference rule that introduced it, so that one SZS status is reported for the whole proof [22] Each rule carries a distinct obligation. An instantiate step is required to be a substitution instance of a parent; a negated_conjecture step to be the negation of its parent, checked by entailment in both directions so that a wrong quantifier dual is detected; a skolemize step to introduce a fresh Skolem symbol that occurs in the result; and a consequence, horn, or deduction step to be entailed by its parents. Entailment is decided by refutation with superposition, resolution, and paramodulation [1]: the parents and the negated conclusion are clausified and saturated, and a derived empty clause certifies the step. The refutation is required to close with the empty clause. Checking is conservative, in keeping with the scoring, under which a wrong verdict is penalised more heavily than abstention: a step is reported unsound only when a saturation is completed that exhibits a counter-model, and a step whose inference rule is not implemented, or whose entailment is neither proved nor refuted within the resource bound, leaves the proof unverified rather than guessed.

Implementation

CheckProof is implemented as a single self-contained C program, with no external dependencies beyond the C standard library. Proofs in TSTP syntax are read directly by a recursive-descent parser into the first-order term representation of the reasoning core. The obligations are discharged by FindProof's own clausal saturation engine, the same system entered in the CASC unit-equality division [10]; no external prover is called, so equational atoms are reasoned about with superposition rather than treated as opaque. The checker is built from source by a single compiler invocation and is available from the author.

Expected Competition Performance

CheckProof is expected to be sound, accepting no invalid proof and rejecting no valid one, and to report a proof unverified when its steps use inference rules outside the implemented set or when an entailment cannot be settled within the time limit.

8.2 GAPT 2.20

Fabian Achammer
TU Wien, Austria

Overview

GAPT (General Architecture for Proof Theory) [4] is a framework containing many data structures, algorithms, parsers and other components common in proof theory and automated deduction. As such it implements various proof calculi, including LK proofs and resolution proofs. The main idea of the

GAPT proof checker is to attempt to import a given TSTP derivation as an acyclic directed graph of LK proofs which can then combined into a compact representation of an LK proof of a sequent whose antecedent contains input axioms and the negation of the input conjecture and whose succeedent contains `$false`. As a pre-pass GAPT performs other auxiliary checks like checking that the axiom file directives are correct. To check semantic entailment of plain inferences GAPT uses its internal superposition prover Escargot.

GAPT returns `VerifiedBad` if any of the checks fail, or any internal proof invariant is violated. If checking takes longer than the given time limit, GAPT returns `Timeout`. If an unexpected situation occurs (e.g., input syntax error, or unexpected exception thrown), GAPT returns `Unknown`. Otherwise, GAPT returns VerifiedGood. The checker not only outputs the SZS status, but in case of `VerifiedBad` it also gives a more detailed message as to what went wrong. For example, if Escargot can establish that a plain inference is incorrect, the checker outputs the step name of the incorrect inference.

The approach of importing the TSTP derivation as an LK proof allows, in principle, to compute a more elaborated version of the TSTP derivation that requires no external ATP tools to check anymore.

Implementation

GAPT is implemented in Scala. It uses parboiled2 as a parsing library to parse the TPTP format. The internal superposition prover Escargot is used for checking plain inferences, so GAPT does not depend on an external ATP tool for proof checking.

The proof checking proceeds in the following way: First the input file is parsed in the TPTP format. Then the file is checked for unique formula names and for acyclicity of the derivation. Next, auxiliary checks are performed to ensure all steps have valid statuses and the file directives in axioms and conjectures are correct. Now, each step is assigned a first-order sequent where the step's parents' formulas are in the antecedent and the step's formula is in the succeedent. For plain inference steps an attempt is made to find an LK proof for this sequent by using the internal superposition prover Escargot. For skolemization inference steps an LK proof for the sequent is constructed using a special skolemization LK rule. For negated conjecture steps an attempt is made to find an LK proof for both the if and only if direction of negation(conjecture) iff negated_conjecture. To make sure that skolem symbols are used appropriately across all inferences, e.g., the same skolem symbol is not used for different skolemized formulas, an additional check is performed across the whole derivation. Finally, for each step, LK cuts between the step's proofs and the step's parents' proofs are applied which results in LK proofs whose antecedents contain axioms and the negation of the conjecture from the input file and whose succeedent contains the formula of the given step. This results in a compact representation of the LK proof of the input refutation, i.e., an LK proof of the sequent whose antecedent consists of input axioms and the negation of the conjecture and whose succeedent consists of `$false`. This representation avoids unfolding the whole LK proof into a tree and instead saves space by remaining a directed acyclic graph.

GAPT is available at:

```
https://www.logic.at/gapt/
```

Expected Competition Performance

We expect GAPT to be sound, but as this is the first version of the GAPT derivation checker we don't expect fast results. It might struggle to verify long derivations or difficult plain inferences, since Escargot works very well on small examples, but is likely not able to compete with state-of-the art resolution provers in this area. However, GAPT should be able to handle most reasonably granular input derivations correctly and we expect to receive few negative points.

8.3 GDV 2.0

Geoff Sutcliffe
University of Miami, USA

Overview

GDV 2.0 [21] is a verifier for derivations in classical first-order and typed first-order logic, written in the TPTP format. GDV checks a derivation in four verification phases: structural verification, leaf verification, rule-specific verification, and inference verification.

- Structural verification deals with non-logical aspects of a proof, including checking the syntax, that formulae are uniquely named, the derivation is acyclic, refutations have *false* roots, etc.
- Leaf verification ensures that the leaves of the derivation match formulae in the original input problem, and that introduced formulae such as definitions meet requirements.
- Rule-specific verification deals with special cases, e.g., splitting as implemented in SPASS [28]. Special techniques are available for verifying correctly documented Skolemization steps; see Section 2.3 of [25].
- Inference verification uses external trusted ATP systems to verify each inference step, based on the SZS status [22] of the inference record. For example, inference steps often have the SZS status `thm` indicating that the inferred formula is (supposed to be) a theorem of the parent formulae. In this case a proof obligation with the parent formulae as the axioms and in the inferred formula as the conjecture is created, and discharged (or not) using a trusted theorem proving ATP system. Other SZS status values are treated with variants of that process.

Implementation

GDV is implemented in C. It is available from:

`https://github.com/TPTPWorld/GDV`

GDV relies heavily on the JJParser library, which has to be downloaded separately into the same directory as GDV:

`https://github.com/TPTPWorld/JJParser`

The external ATP systems run remotely, through the SystemOnTPTP service [20].

Expected Competition Performance

The short time limit of 30s per proof, and the relatively slow process of running the remote ATP systems, means that GDV is likely to timeout often. GDV is sound, slow, but very sure of itself.

8.4 GDV-LP 2.0

Frédéric Blanqui
ENS Paris-Saclay, INRIA, France

Overview

GDV-LP 2.0 [25] is a verifier for derivations in classical first-order and typed first-order logic, written in the TPTP format. GDV-LP checks a derivation in two steps:

- Standard GDV (as described in section 8.3 above) is run, using ZenonModulo [3] as the trusted ATP system for discharging proof obligations. ZenonModulo is configured to output a LambdaPi term [6] for each discharge proof.

- GDV-LP produces the necessary files that declare the formulae, the signatures of the symbols in the terms, a LambdaPi term for the root of the proof, and a `lambdapi.pkg` package file. ZenonModulo's LambdaPi terms are chained together from the root term, and passed to the `lambdapi` checker to be checked. The key strength of this added layer is that it is not necessary to trust the "trusted ATP system", here ZenonModulo. Additionally, tools other than `lambdapi` can be used to check the LambdaPi terms, e.g., `dkcheck` [12] and `kontroli` [5].

Implementation

GDV-LP is implemented in C. It is available from:

`https://github.com/TPTPWorld/GDV`

GDV relies heavily on the JJParser library, which has to be downloaded separately into the same directory as GDV:

`https://github.com/TPTPWorld/JJParser`

ZenonModulo is implemented in OCaml. It is available from:

`https://github.com/Deducteam/zenon\_modulo`

`lambdapi` is written in OCaml. It is available from:

`https://github.com/Deducteam/lambdapi.git`

ZenonModulo, other external ATP systems, and `lambdapi` run remotely, through the SystemOnTPTP service [20].

Expected Competition Performance

The short time limit of 30s per proof, and the relatively slow process of running the remote ATP systems, means that GDV-LP is likely to timeout often, probably in the first non-LambdaPi step. GDV-LP is sound, slow, but very very sure of itself.

8.5 mrs-proover 0.2.0

Olivier Roland
Independent Researcher, France

Architecture

mrs-proover 0.2.0is a proof checker for TSTP first-order refutation proofs, built as a companion to the mrs automated theorem prover. It follows the semantic verification paradigm pioneered by GDV [21]: structural properties of the proof DAG are checked first (uniqueness of formula names, acyclicity, a single `$false` root), followed by leaf verification against the linked problem file, followed by per-step verification of each inference.

A small set of inference patterns that are expected to recur in every proof — the negated-conjecture step, and Skolemization steps — are verified internally using dedicated structural checks rather than external provers, since these steps have precisely specified shapes (see the ProoVer Rules and Format page). Axiom and conjecture leaves are checked for alpha-equivalence against the named formula in the linked problem file, either internally or, when internal matching fails, by delegating to an external ATP. All other inference steps with status `thm` or `cth` are discharged as proof obligations to an ATP ladder.

mrs-proover reports `Unknown` (rather than guessing) whenever neither a positive nor a negative verdict can be established within the allotted resources for a given step; because the ProoVer scoring penalizes a false `VerifiedGood` on a bad proof ten times more heavily than a missed detection, the overall verdict policy is deliberately conservative: any single step found unsound anywhere in the proof yields `VerifiedBad` for the whole proof; otherwise any step left undecided yields `Unknown`; only if every step is positively confirmed does the proof yield `VerifiedGood`.

Implementation

mrs-proover is implemented in Rust (edition 2024), reusing the mrs TPTP/TSTP parser, clausifier, and unification/formula-lowering crates from the same workspace. The proof is loaded and its dependency structure is represented as a directed acyclic graph over annotated formulae; formulae are lowered from the TPTP AST into the shared mrs-core term/formula representation for alpha-equivalence and structural comparisons. For inference steps that cannot be settled by the internal structural checks, mrs-proover discharges the corresponding proof obligation (parent formulae as axioms, inferred formula as conjecture) to an ATP ladder that runs, per step, the in-process mrs prover first, then races any available external backends — E [13] and Vampire [7] — in parallel; the first definite Sound/Unsound verdict from any ladder member wins and the remaining subprocesses for that step are cancelled. Steps can be verified concurrently across multiple CPU cores (default 8, matching the competition hardware).

mrs-proover has been cross-checked, using only its internal mrs backend (no external ATPs), against the 7 official ProoVer example proofs, against the leoprover/noergler PyRes correctness corpus (170 valid proofs, 170 corresponding falsified/mutated proofs), and against the ATP-Research-Project test corpus of correct and deliberately "evil" proofs. Across all of these, running the in-process backend alone never produced a false `VerifiedGood` on an evil proof and never produced a false `VerifiedBad` on a valid proof; adding the external E/Vampire backends only improves detection rate (fewer proofs left at `Unknown`), never soundness.

mrs-proover is open source (MIT OR Apache-2.0) and available from:

`https://github.com/newca12/mrs`

as the mrs-proover crate in the same repository as the mrs ATP system.

Expected Competition Performance

mrs-proover is expected to correctly identify almost all well-formed valid proofs as `VerifiedGood` and almost all deliberately incorrect ("evil") proofs as `VerifiedBad`, given the internal structural checks tailored to the exact Skolemization and negated-conjecture formats specified by the ProoVer rules, backed by external ATP calls for the remaining steps. Given the highly asymmetric scoring (a single false Verified on an evil proof costs as much as ten correct verifications), mrs-proover is tuned to prefer reporting `Unknown` over guessing whenever a step cannot be positively confirmed or refuted within the time budget, so a non-trivial fraction of harder or unusually-shaped valid proofs may be conservatively scored 0 rather than +1.

8.6 Nörgler 1.1

Alexander Steen
University of Greifswald, Germany

Overview

The proof checker Nörgler 1.1 [27, 18]. is designed as a light-weight system for the verification of TSTP refutations across various logic dialects, including propositional, untyped or typed first-order, and higher-order logics. The architecture is built upon the semantic verification paradigm pioneered

by the GDV system [21]. This approach combines structural property verification using dedicated checks with the verification of individual proof steps by re-proving each inference using trusted general-purpose automated theorem provers. A core design choice in the system is the integration of multi-core parallelization, which allows independent proof checking tasks to be processed concurrently, thereby significantly increasing performance and reducing overall verification time.

The verification process is executed via a multi-stage checking strategy. Initially, global structural properties of the proof (such as ensuring the proof graph is acyclic, verifying that formula names are unique, and confirming that the derivation correctly terminates in false) are validated. Following these global assessments, individual logical inferences are evaluated. This comprises checks of local structural properties (such as ensuring that a given step has an appropriate TSTP role and that all listed parents exist earlier in the proof) and the semantical verification of the inferences. Nörgler checks the faithfulness of all formulae taken from the problem file. With the exception of the negation of the conjecture and Skolemisazation, which are verified by dedicated checks, the inferences are verified by invoking an external trusted prover. If this check does not succeed within the allocated resource limit, countermodel search is invoked via integrated model finders to actively detect incorrect proof steps. If a countermodel is found, the inference is rejected as invalid (resulting in a `VerifiedBad` status); however, if neither a successful proof nor a definitive countermodel can be established by the auxiliary tools, a status of `Unknown` is reported. If all checks are successful, Nörgler returns `VerifiedGood`.

Several notable features are incorporated into the system to enhance its flexibility and robustness. Fine-grained control over the strictness with which formatting conventions are enforced is provided, enabling the system to either gracefully handle imperfect prover outputs or strictly demand adherence to proof standards, see [27] for details.

Implementation

Nörgler is implemented in Scala, making use of its native support for concurrency via Futures and functional data structures. The system incorporates the `scala-tptp-parser` library [17] for robust parsing of the standard TPTP/TSTP grammar formats, and integrates the `tptp-utils` tool for formula manipulation [16]. For its backend verification tasks, the well-known systems E [13] and Mace4 [9] (for proving resp. disproving) are used. The system is freely available as open-source software (MIT license), and can be accessed online via its repository at GitHub:

`https://github.com/leoprover/noergler/`

Expected Competition Performance

This is the very first version of Nörgler, so no reliable expectations can be formulated. The author's are positive, however, that Nörgler should not report false positives (i.e., reporting a successful verification on a flawed proof).

8.7 ProofCheck 1.0

Jeff Machado
Independent Researcher, USA

Overview

ProofCheck 1.0 is a verifier for first-order refutation proofs written in the TPTP format. A proof is checked by structural verification followed by independent verification of every inference step, and is reported as `VerifiedGood`, `VerifiedBad`, `Unknown`, or `Timeout`. The verifier has a single, strict mode: by construction, no derived step can be accepted without verification, and a step that cannot be decided is reported as `Unknown` rather than `VerifiedGood`, to avoid false acceptance.

Structural verification checks the syntax, unique naming, acyclicity, a `$false` root, and that the leaves match the input problem (located from the proof's `% Proof :` header). Leaf bodies are compared against their cited problem formulae canonically, with an ATP equivalence check as the fallback, and clausal leaves are validated by replaying the clausification of their source formula. Introduced definitions are checked for conservativity and Skolem-symbol laundering.

Inference verification discharges every `thm` and `cth` step by an ATP proof obligation built from exactly that step and its cited premises: the parents as axioms and the conclusion as conjecture, discharged by a local E [14] raced against Mace4 [8], where a Mace4 countermodel refutes the step; a step that E leaves undecided is retried with Vampire [7] as a second entailment oracle before the step is hedged. SideStep, a built-in clausal inference engine, replays clausal steps structurally in process first; a step it refutes is rejected outright, while its acceptances are not trusted and the step still goes to the ATP. Skolemization (`esa`) steps are never sent to an ATP; they are verified structurally against the parent: the required `skolemize(Var,Term)` record, the Skolem term's arguments against the universally quantified variables in scope at the eliminated existential, symbol freshness across the proof and against the problem's own symbols, and one fresh symbol per eliminated (non-vacuous) existential. Constructs outside the specification are hedged to Unknown. A wall-clock deadline is self-imposed, so one of the four verdicts is always emitted within the time budget.

Implementation

ProofCheck is implemented in C++, including the SideStep engine and the structural checkers. It compiles into a single static binary that invokes E [14] and Vampire [7] as entailment provers, Mace4 [8], as the countermodel finder, and Prover9 [8], for clausification replay, all locally from the directory of the binary. It is available from:

`https://github.com/AlgorithmicTruth/proofcheck-releases`

Expected Competition Performance

The entailment backends are local and most structural checks are in process, so ProofCheck verifies typical proofs in well under a second and rarely times out. It has been tested for soundness on an adversarial library of over 200 red-team proofs, a third-party evil proof suite, and several hundred mutated E proofs, with no false `VerifiedGood` observed, and for coverage on several hundred valid E and Prover9 proofs. Constructs the verifier does not model are reported as `Unknown` rather than accepted, so unanticipated cases reduce coverage, not soundness. ProofCheck is sound and fast.

8.8 ProofGuard—1.0

Matthew Farah
McMaster University, Canada

Overview

ProofGuard 1.0 is an automatic checker for first-order logic proof certificates submitted in the TSTP format [24]. A problem file and a proposed proof file are taken as input, and it is checked whether the proof establishes the claimed contradiction. The system is designed for refutation proofs. A proof is accepted only when every step of the argument can be followed and it can be confirmed that `$false` is correctly derived. When all required steps have been verified, the proof is reported as verified. ProofGuard is intentionally conservative. When a proof step is unclear, unsupported, malformed, or cannot be checked within the available time, no guess is made. It is instead reported that the proof could not be verified. A proof is rejected as flawed only when a definite error can be identified, such as an incorrect negated conjecture, an invalid Skolemization step, or an inference which does not follow from its stated premises. This design reflects the competition setting, in which the incorrect acceptance

of a bad proof is heavily penalized. Caution is therefore prioritized, wherein only fully checked proofs are accepted, only demonstrably flawed proofs are rejected, and verification is otherwise reported as inconclusive. Verdicts are reported using the required SZS statuses [22]. An accepted proof is reported as `VerifiedGood`; a rejected proof is reported as `VerifiedBad`, together with a one-line reason naming the flawed step; an inconclusive verification is reported as `Unknown`; and an exhausted time budget is reported as `Timeout`.

Implementation

ProofGuard 1.0 (July 2026) is the first release of the system. The E prover, built from current upstream sources, is bundled with the system as the backend ATP.

- Problem and proof parsing. FOF problem files and TSTP proof files are parsed by a recursive-descent parser, and formulas are represented as immutable dataclass terms. Nested inference(...) chains are flattened to their leaf premises.
- Input-step validation. Steps imported from the problem file are checked against the referenced problem formula, modulo alpha-renaming and harmless syntactic normalizations such as literal order and equation orientation.
- Internal rule checking. Specified rules are verified internally. Negated-conjecture steps are checked against the negation of the original conjecture, which may be consumed by no other step. Skolemization steps are checked for fresh Skolem symbols, consistent annotations, and correct dependency on the universal variables in scope.
- Conservative de-Skolemization fallback. When Skolemization annotations are absent, de-Skolemization is attempted only when the transformation is unambiguous: each fresh Skolem term is replaced by a new existentially bound variable, after which the remaining theorem-status content is checked by the ATP. Whenever the transformation cannot be performed unambiguously, verification is reported as inconclusive.
- ATP-backed consequence checking. Unspecified inference steps marked with theorem status are checked through the generation of local proof obligations for the E prover [14]. The original conjecture is never independently re-proved.
- Conservative failure policy. A demonstrably incorrect step causes a failed verification to be reported. A malformed, unsupported, ambiguous, timed-out, or internally failing check causes verification to be reported as inconclusive; the system is never crashed by malformed input.
- Resource management. A global wall-clock budget is managed by the checker. Each ATP call is given a CPU limit derived from the remaining time, and the system is terminated conservatively before the external time limit is exceeded.
- Packaging and reproducibility. ProofGuard is implemented in pure Python 3 with no third-party Python libraries. The delivery package is built with Docker and is fully self-contained: the CPython runtime, the E binary, and the checker are all included, glibc 2.17 is targeted, and the system is launched as ./proover-check PROBLEM PROOF.
- Testing. The system was tested on the published ProoVer examples, on E-generated TSTP proofs over the TPTP library, on multi-system TSTP samples, and on mutation-fuzzed proof variants.

The source is available here:

`https://github.com/ValueAchooMatthew/ATP-Research-Project`

Expected Competition Performance

All published example problems are classified correctly, and each evil proof is rejected for its intended flaw. Wrong answers are expected to be rare: across 58 TSTP proofs produced by five different systems, no correct proof was misclassified as flawed, and no mutation-fuzzed proof was ever accepted. The strongest performance is expected on invalid proofs, for which structural checks and countersatisfiability results are obtained cheaply and definitively. On the published 8 MB stress sample, a flaw

(a missing Skolem dependency at step 219 of 223) is located in roughly 12 seconds, without any ATP call on the large formulas. Valid proofs written in the style of the published examples are expected to be verified within the time limit, while proofs relying on techniques outside the system's coverage are reported as inconclusive rather than risked. Overall, it is expected that most lost points will be attributable to inconclusive verdicts rather than to penalties.

8.9 PyCheck 0.1

Stephan Schulz
DHBW Stuttgart, Germany

Overview

PyCheck is a simple (mostly) semantic proof checker for refutation-based proofs in the most commonly used subset of TPTP/TSTP CNF/FOF syntax. Inspired by GDV [21], it performs both structural and semantic checks on proof files. Among the structural checks are existance of an explicit contradiction, acyclicity of the proof graph, and freshness of introduced (Skolem) symbols. Semantic checks include verification of "status(thm)" and "status(cth)" checks using the external prover.

Implementation

PyCheck reuses the parser and a lot of internal data structures from PyRes [15], and thus supports the same core of the TPTP-3 proof language, with support added for more detailed derivations.

PyCheck calls an external prover (usually E [14]) to verify most proof steps (in particularly those with `thm` and `cth` semantics). It uses internal processing to try to verify Skolemization steps, to check existence of axioms in the original input file, and to perform structural checks on the proof graph.

PyCheck can be downloaded from GitHub:

```
https://github.com/eprover/PyCheck
```

It requires E (`https://www.eprover.org` to be available in its search path (the ProoVer StaExec package should automatically do everything necessary).

Expected Competition Performance

PyCheck 0.1 is at a very early state of maturity. It can handle all the problems given in the ProoVer description, but may fail to catch some particulary malicious structural problems of proofs. Overall, it should prove to be at least useful.

8.10 VaLeaDate 0.1

Jonas Bodingbauer
TU Wien, Austria

Overview

VaLeaDate v0.1 is a proof verification system for TPTP proofs. It leverages a recently developed proof output from Vampire, which generates proofs as Lean input files that are end-to-end verifiable by Lean [2]. The core approach of VaLeaDate is to invoke Vampire on each inference, inspired by GDV [21], chain the resulting proof steps together, and subsequently verify the entire proof in Lean. Since the proof checking step is time-consuming, VaLeaDate checks multiple structural properties of the proof in the beginning and terminates early if any of these checks fail. These basic checks include:

- Verifying that all parent nodes exist
- Checking for acyclicity of the proof structure
- Ensuring axioms are alpha-equivalent to the problem input

After that, all inferences are passed to the ATP Vampire, which processes them in a parallelized manner. The resulting proof steps are then chained together (in one or multiple files depending on the size of the proof output) and verified by Lean. Skolemization is handled within VaLeaDate, generating a skolemization section in the Lean proof output, which is then verified by Lean. The process transforms any input into NNF, skolemizes it, and then reconstructs it (if necessary) back to the expected form.

The system reports status `Unknown` for (some) syntactic errors or if Vampire does not produce a conclusive result (neither satisfiable nor a refutation). `VerifiedBad` is produced if any of the basic checks fail or if the Lean verification fails. Finally, `VerifiedGood` is produced if all checks pass and the final Lean verification also succeeds.

Implementation

VaLeaDate is implemented in Scala and utilizes the scala-tptp-parser parser from the leoprover project to parse TPTP input and proof files. Furthermore, it relies on Lean, a small Lean library VampLean, as well as a custom Vampire build configured for proof output (including VaLeaDate-specific functions). It aims to parallelize the computationally intensive parts of proof reconstruction and verification as much as possible.

Expected Competition Performance

It is challenging to predict the performance, but it is hoped that VaLeaDate does not produce any false positives. They can still occur due to the somewhat open definition of correctness or other bugs introduced during the relatively short development period. Since verification in Lean introduces some overhead, the system might time out on many problems within the given 30s time limit.

9 Conclusion

ProoVer 2026 was the first edition of the Proof Verification Competition. The competition organizers believe that ProoVer fulfills its main motivations: evaluation of the relative capabilities of proof checking systems, stimulation of research into the definition of what a should be proof, an occasion for the community to discuss standards for proofs, and the provision of an exciting event, run alongside CASC at IJCAR/FLoC.

Through the continuity of the event, and consistency in the reporting of results, performance comparisons with future editions will become possible as ProoVer establishes itself.

The competition provides exposure for proof checker builders both within and outside of the automated deduction community, and gives an overview of the state of the art in checking first-order proofs expressed in the TSTP format.

Acknowledgement. The organizers warmly thank the members of the ProoVer 2026 panel (Cláudia Nalon, Aart Middeldorp, and Martina Seidl), and especially Geoff Sutcliffe, the organizer of CASC, for his invaluable advice and for managing its infrastructure. The present proceedings are largely inspired by the yearly CASC proceedings.

References

[1] L. Bachmair and H. Ganzinger. Rewrite-based Equational Theorem Proving with Selection and Simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.

[2] J. Bodingbauer and M. Hajdu. Lean on Vampire Proofs. In E. Komendantskaya, editor, *Proceedings of the 17th Conference on Interactive Theorem Proving*, page To appear. Leibniz International Proceedings in Informatics, 2026.

[3] D. Delahaye, D. Doligez, F. Gibert, P. Halmagrand, and O. Hermant. Zenon Modulo: When Achilles Outruns the Tortoise using Deduction Modulo. In K. McMillan, A. Middeldorp, and A. Voronkov, editors, *Proceedings of the 19th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, number 8312 in Lecture Notes in Computer Science, pages 274–290, Berlin, Germany, 2013. Springer.

[4] G. Ebner, S. Hetzl, G. Reis, M. Riener, S. Wolfsteiner, and S. Zivota. System Description: GAPT 2.0. In N. Olivetti and A. Tiwari, editors, *Proceedings of the 8th International Joint Conference on Automated Reasoning*, number 9706 in Lecture Notes in Artificial Intelligence, pages 293–301, 2016.

[5] M. Färber. Safe, Fast, Concurrent Proof Checking for the lambda-pi Calculus Modulo Rewriting. In A. Popescu and S. Zdancewic, editors, *Proceedings of the 11th International Conference on Certified Programs and Proofs*, pages 225–238. Association for Computing Machinery, 2022.

[6] G. Hondet and F. Blanqui. Encoding of Predicate Subtyping and Proof Irrelevance in the λΠ-calculus Modulo Theory. In U. de'Liguoro, S. Berardi, and T. Altenkirch, editors, *Proceedings of the 26th International Conference on Types for Proofs and Programs*, number 188 in Leibniz International Proceedings in Informatics, pages 6:1–6:18. Dagstuhl Publishing, 2020.

[7] L. Kovács and A. Voronkov. First-Order Theorem Proving and Vampire. In N. Sharygina and H. Veith, editors, *Proceedings of the 25th International Conference on Computer Aided Verification*, number 8044 in Lecture Notes in Artificial Intelligence, pages 1–35, Berlin, Germany, 2013. Springer.

[8] J.P. Machado and L. Lesyna. Enhanced LADR-2026. https://prover9.org.

[9] W.W. McCune. Mace4 Reference Manual and Guide. Technical Report ANL/MCS-TM-264, Argonne National Laboratory, Argonne, USA, 2003.

[10] N. Murzin. FindProof: Equational Theorem Proving in the Wolfram Language. Wolfram Institute Technical Note, 2026.

[11] O. Roussel. Controlling a Solver Execution with the `runsolver` Tool. *Journal of Satisfiability, Boolean Modeling and Computation*, 7(4):139–144, 2011.

[12] R. Saillard. *Typechecking in the lambda-Pi-Calculus Modulo : Theory and Practice*. PhD thesis, Ecole Nationale Supérieure des Mines de Paris, Paris, France, 2015.

[13] S. Schulz. E: A Brainiac Theorem Prover. *AI Communications*, 15(2-3):111–126, 2002.

[14] S. Schulz, S. Cruanes, and P. Vukmirović. Faster, Higher, Stronger: E 2.3. In P. Fontaine, editor, *Proceedings of the 27th International Conference on Automated Deduction*, number 11716 in Lecture Notes in Computer Science, pages 495–507, Berlin, Germany, 2019. Springer.

[15] S. Schulz and A. Pease. Teaching Automated Theorem Proving by Example: PyRes 1.2 (system description). In N. Peltier and V. Sofronie-Stokkermans, editors, *Proceedings of the 10th International Joint Conference on Automated Reasoning*, number 12167 in Lecture Notes in Computer Science, pages 158–166, 2020.

[16] A. Steen. tptp-utils v1.4.2, 2025.

[17] A. Steen. Scala TPTP Parser v1.7.4, 2026.

[18] A. Steen, N. Taprogge, and H. Sariyanto. Nörgler, 2026.

[19] A. Stump, G. Sutcliffe, and C. Tinelli. StarExec: a Cross-Community Infrastructure for Logic Solving. In S. Demri, D. Kapur, and C. Weidenbach, editors, *Proceedings of the 7th International Joint Conference on Automated Reasoning*, number 8562 in Lecture Notes in Artificial Intelligence, pages 367–373, 2014.

[20] G. Sutcliffe. SystemOnTPTP. In D. McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction*, number 1831 in Lecture Notes in Artificial Intelligence,

pages 406–410, Berlin, Germany, 2000. Springer.

[21] G. Sutcliffe. Semantic Derivation Verification: Techniques and Implementation. *International Journal on Artificial Intelligence Tools*, 15(6):1053–1070, 2006.

[22] G. Sutcliffe. The SZS Ontologies for Automated Reasoning Software. In G. Sutcliffe, P. Rudnicki, R. Schmidt, B. Konev, and S. Schulz, editors, *Proceedings of the LPAR Workshops: Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th International Workshop on the Implementation of Logics*, number 418 in CEUR Workshop Proceedings, pages 38–49, 2008.

[23] G. Sutcliffe. The CADE ATP System Competition - CASC. *AI Magazine*, 37(2):99–101, 2016.

[24] G. Sutcliffe. Stepping Stones in the TPTP World. In C. Benzmüller, M. Heule, and R. Schmidt, editors, *Proceedings of the 12th International Joint Conference on Automated Reasoning*, number 14739 in Lecture Notes in Artificial Intelligence, pages 30–50, 2024.

[25] G. Sutcliffe, F. Blanqui, and G. Burel. Proof Verification with GDV and LambdaPi - It's a Matter of Trust. In I. Biskri and D. Talbert, editors, *Proceedings of the 38th International FLAIRS Conference*, 2025.

[26] G. Sutcliffe, S. Schulz, K. Claessen, and A. Van Gelder. Using the TPTP Language for Writing Derivations and Finite Interpretations. In U. Furbach and N. Shankar, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, number 4130 in Lecture Notes in Artificial Intelligence, pages 67–81, Berlin, Germany, 2006. Springer.

[27] M. Taprogge, H. Sariyanto, and A. Steen. A Light-weight Proof Checker for TSTP Refutations. In C. Nalon, M. Hajdu, and M. Suda, editors, *Proceedings of the 10th Workshop on Practical Aspects of Automated Reasoning*, CEUR Workshop Proceedings, page To appear, 2026.

[28] C. Weidenbach. Combining Superposition, Sorts and Splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 1965–2011. Elsevier Science, Amsterdam, The Netherlands, 2001.