# Proof Verification with GDV and LambdaPi
# It's a Matter of Trust

**Geoff Sutcliffe**
University of Miami
Miami, USA

**Frédéric Blanqui**
Université Paris-Saclay, ENS Paris-Saclay,
CNRS, **INRIA**, Laboratoire Méthodes Formelles
Gif-sur-Yvette, France

**Guillaume Burel**
Laboratoire Samovar
École Nationale Supérieure
d'Informatique pour l'Industrie et l'Entreprise
Évry-Courcouronnes, France

### Abstract

Automated Theorem Proving (ATP) is concerned with the development and use of software that automates sound reasoning. An ATP system can be required to output a proof that serves as a certificate for the system's claim. To ensure that a proof is correct, verification can be required. If the verifier outputs evidence in a form that can be independently checked, that evidence serves as a certificate for the verifier's claim. The sequence of finding a proof, verifying the proof, and certifying the verification, builds an increasing level of trust in the system. This paper traces one such path for TPTP format proofs generated by ATP systems, via the GDV derivation verifier, and ending at the LambdaPi checker.

## 1 Introduction

Automated Theorem Proving (ATP) (Robinson and Voronkov 2001) is concerned with the development and use of software that automates sound reasoning: the derivation of conclusions that follow inevitably from known facts. ATP is at the heart of many computational tasks, including sensitive tasks such as software/hardware verification (Hähnle and Huisman 2019) and system security (Cook 2018). ATP systems are often used as components of more complex Artificial Intelligence (AI) systems, which means that the impact of ATP extends into many facets of society. In many of these applications the use of ATP systems is mission critical, in the sense that incorrect results from ATP might have nasty consequences. The importance of verifying the results from autonomous systems (including ATP systems) is reflected in the IEEE P2817 standard, which aims to "identify best practices and provide guidance that supports the definition of valid verification processes for a range of autonomous system configurations".[1]

Facing the demand for error-free results from ATP systems is the reality that ATP systems are complex pieces of software, implementing complex calculi with complex data structures and algorithms (Schulz 2006). Despite best intentions and efforts, incorrect results are possible. To counter incorrectness, an ATP system can be required to output a proof that serves as a certificate for the system's claim. To ensure that a proof is correct, proof verification can be required, which serves as a certification (but not a certificate) of the proof. If the verifier outputs evidence for the certification in a form that can be independently checked, that evidence serves as a certificate for the verifier's claim. As a concrete example, consider the verification process for aerospace software, shown in Figure 1, taken from (Sutcliffe, Denney, and Fischer 2005). The "proofs" output by the ATP system are certificates that the "safety policy" has been verified. However, certification authorities like the FAA must be given explicit evidence that the individual tool components (here, the "ATP" system) yield correct results. To that end the ATP system's proofs are given to a "proof checker" that produces certificates that are attached to the "code".

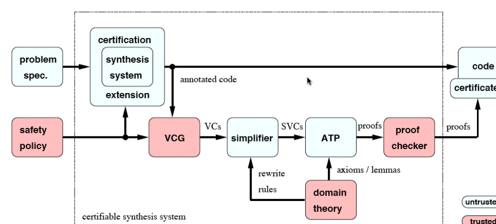


Figure 1: Practical proof checking for program certification

The sequence of finding a proof, verifying the proof, and certifying the verification, builds an increasing level of trust in the system. Each step is based on a lack of trust in the preceding software, and greater trust in the succeeding software. This paper traces one such path for TPTP format proofs generated by ATP systems (Sutcliffe et al. 2006), via the GDV derivation verifier (Sutcliffe 2006), and ending at the LambdaPi checker (Hondet and Blanqui 2020).

### 1.1 Verification and Trust

Proof verification can be viewed incrementally:

1. Proofs must be parsible in an agreed language. The TPTP language (Sutcliffe 2023) is appropriate. A parser based on the BNF definition of the TPTP language (Van Gelder and Sutcliffe 2006) can be used to check conformance.

Copyright © 2025 by the authors.
This open access article is published under the Creative Commons Attribution-NonCommercial 4.0 International License.

[1] standards.ieee.org/ieee/2817/11726/

2. Proofs must be written in an agreed upon concrete format using the agreed language. The established TPTP format for proofs (Sutcliffe et al. 2006) has already been adopted by many ATP systems.
3. Proofs must be structurally correct. The parents of inferred formula must be documented and exist in the derivation, the derivation must be acyclic, refutations must have *false* roots, assumptions must be discharged, etc. GDV can be used to check conformance.
4. Proofs must be for the given problem. The leaves of a derivation must come from the problem.
5. Proofs must be logically complete and correct. This is where most attention has been focussed in proof verification, (often at the expense of the preceeding requirements that are simply assumed). Proof verification is well researched, with multiple approaches, including formal system development, e.g., (Schlichtkrull et al. 2020), empirical observation, the LCF philosophy, e.g., (Gordon, Milner, and Wadsworth 1979), proof replay, e.g., (Andreotti, Lachnitt, and Barbosa 2023), higher-order techniques, e.g., (Harper, Honsell, and Plotkin 1993), and semantic verification, e.g., (Sutcliffe 2006; Ebner et al. 2016). There are several proof verification systems that can be used, including GDV (Sutcliffe 2006), DEDUKTI (Saillard 2015), and GAPT (Ebner et al. 2016).

Empirical testing, as is done by most ATP system developers, provides a reasonable assurance that an ATP system outputs syntactically well-formed proofs, but is only a step towards trusting an ATP system. Formal verification of an ATP system's proofs provides evidence of logical correctness. From a user perspective, in addition to being well-formed and logically correct, proofs must be comprehensible to the applications (including humans) that need to use the proofs (Reger 2016). ATP systems that pass verification tests consistently over time become trusted, and trusted proofs that are comprehensible become useful, in the process shown in Figure 2. The sequence of events (typically over years) is ...

1. Problems are given to an untrusted ATP system, which produces untrusted proofs.
2. The untrusted proofs are checked against expectations and other ATP systems' results, e.g., if a problem is expected to be a theorem and other trusted ATP systems have reported that it is a theorem, hopefully the untrusted system will agree with that.
3. The form of the untrusted proofs is checked.
4. If the proofs are well-formed they can be logically verified.
5. The results of steps 1-4 contribute to an accumulation of evidence about the ATP system, which induces a level of trust in the system.
6. After enough positive evidence has accumulated the ATP system becomes trusted (this is a socio-empirical process) and its proofs are trusted, e.g., Otter (McCune 2003) is commonly trusted, thanks to its extensive usage by many researchers over many years. The decision to trust an ATP system might be formalized in a framework such as the Distributed Assertion Management Framework (Al Wardani, Chaudhuri, and Miller 2023).
7. Trusted proofs can still be subject to the checks of steps 2-4, to further impact the level of trust in the ATP system.
8. If trusted/verified proofs are (human) comprehensible they are useful to (human) applications.

This paper describes work done to verify proofs output by ATP systems so that (i) the user has verified proofs, (ii) over time the user builds trust in the ATP system, and finally (iii) the user has useful proofs.

# 2 Background

## 2.1 TPTP Derivations

The TPTP language (Sutcliffe 2023) is one of the keys to the success of the TPTP World. The TPTP language is used for writing both problems and solutions, which enables convenient communication between ATP systems and tools. Problems and solutions are built from *annotated formulae* of the form:

*language*(*name*, *role*, *formula*, *source*, *useful_info*)

The supported *language*s are `cnf` (clause normal form), `fof` (first-order form), `tff` (typed first-order form), and `thf` (typed higher-order form). The *role*, e.g., `axiom`, `lemma`, `conjecture`, defines the use of the formula. The logical connectives in the TPTP language are `!`, `?`, `~`, `|`, `&`, `=>`, `<=`, `<=>`, and `<~>`, for the mathematical connectives $\forall$, $\exists$, $\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\Leftarrow$, $\Leftrightarrow$, and $\oplus$ respectively. The *source* and *useful_info* are optional. Figure 3 shows a typed first-order form problem.

A derivation written in the TPTP language is a list of annotated formulae. The role for leaves is typically one of `axiom` or `conjecture`, and the role for inferred formulae is typically one of `negated_conjecture` (also used for leaves in CNF) or `plain` for inferred formulae. The source is either a `file` record for leaves or an `inference` record for inferred formulae. A `file` record contains the problem file name and the corresponding annotated formulae name in the problem file. An `inference` record contains the inference rule name, a list of useful inference information, and a list of the parent formulae. The parent formulae list can contain parent annotated formulae names, and nested `inference` records. Common types of useful inference information are the semantic relationship of the inferred formula to its parents as an SZS ontology value (Sutcliffe 2008) in a `status` record, special information about recognized types of complex inference rules, e.g., Skolemization and explicit splitting, and details of new symbols introduced in the inference.

The use of SZS values is core to GDV's approach to verification, explained in Section 2.2. The SZS ontologies (Sutcliffe 2008) provide values to specify the logical status of problems and solutions, and to describe logical data. The Success ontology is relevant here – it provides values for the logical status of a conjecture with respect to a set of axioms, e.g., a TPTP problem whose conjecture is a logical consequence of the axioms is tagged as a `Theorem`, and a model finder that establishes that a set of axioms (with no conjecture) is consistent should report `Satisfiable`. The Success ontology is also used to specify the semantic relationship between the parents and inferred formulae of an inference. The SZS values that are used in this work are shown in Table 1.

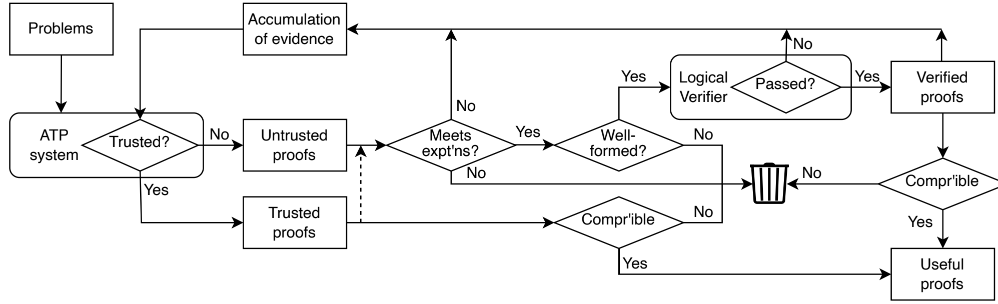


Figure 2: The cycle of trust

| | | |
|---|---|---|
| SUC | Success | Data has been processed successfully. |
| ESA | EquiSatisfiable | There exists a model of the axioms iff there exists a model of the conjecture. |
| SAT | Satisfiable | Some interpretations are models of the axioms. |
| EQV | Equivalent | The axioms and conjecture have the same models. |
| THM | Theorem | All models of the axioms are models of the conjecture. |
| CTH | CounterTheorem | All models of the axioms are models of the negated conjecture. |
| CAX | ContradictoryAxioms | No interpretations are models of the axioms. |
| ECS | EquiCounterSatisfiable | There exists a model of the axioms iff there exists a model of the negated conjecture. |
| CSA | CounterSatisfiable | Some models of the axioms are models of the negated conjecture. |
| CEQ | CounterEquivalent | The axioms and negated conjecture have the same models. |

Table 1: SZS ontology values

```
%------------------------------------------------------------------
tff(human_decl,type,human: $tType).
tff(grade_decl,type,grade: $tType).
tff(john_decl,type,john: human).
tff(a_decl,type,a: grade).
tff(f_decl,type,f: grade).
tff(grade_of_decl,type,grade_of: human > grade).
tff(created_equal_decl,type,created_equal: (human * human) > $o).

tff(all_created_equal,axiom,
    ! [H1:human,H2:human] : created_equal(H1,H2) ).

tff(john_got_an_f,axiom,
    grade_of(john) = f ).

tff(someone_got_an_a,axiom,
    ! [O: human] : ? [H:human] : ( O != H & grade_of(H) = a ) ).

tff(f_is_not_a,axiom,
    f != a ).

tff(there_is_someone_else,conjecture,
    ? [H:human] : ( H != john & created_equal(H,john) ) ).
%------------------------------------------------------------------
```

Figure 3: Example problem

Figure 4 shows a proof for the problem in Figure 3. There are some points salient to verification:

- All the leaves except ax4 are copies of formulae in the problem. The leaf ax4 is the symmetric version of f_is_not_a in the problem.
- Many of the formulae are inferred with SZS status THM, i.e., they are logical consequences of their parents.
- The negated conjecture inf1 is inferred with SZS status CTH, its negation is a logical consequence of its parent.
- The Skolemized formula inf4 is inferred with SZS status ESA. Note the information about the Skolem symbol in the new_symbols record, and the Skolemized variable in the skolemized record.
- Two of the inferred formulae, inf5 and proof, have nested inference records, and the intermediate inferred formula has not been recorded in the derivation. For example, in proof the nested inference between inf3 and ax2 probably produced ! [X1: human] : grade_of(X1) = f (but it could have been the symmetric version of that).

## 2.2 The GDV Derivation Verifier

The GDV derivation verifier (Sutcliffe 2006) was developed at the start of the century, primarily targeting proofs by refutation in clause normal and first-order form. Over time it has been incrementally developed to verify proofs in typed first-order form and typed higher-order form. GDV's input is a TPTP format proof, and optionally (required for complete verification) the problem for which the proof was produced.

GDV checks a TPTP proof in four verification phases: structural verification, leaf verification, rule-specific verification, and inference verification. In the details below, failed checks that indicate an error in the proof are tagged "(E)" for "error". Failed checks that don't indicate an error in the proof, but warrant closer inspection, are tagged "(W)" for "warning".

Many of the checks rely on a "check-by-ATP", which calls a trusted ATP system – either a theorem prover or a model finder. In all calls to a theorem prover, a model finder is used to check-by-ATP if the axioms of the check are sat-

```
%------------------------------------------------------------------
tff(human_type,type,human: $tType).
tff(grade_type,type,grade: $tType).
tff(john_decl,type,john: human).
tff(a_decl,type,a: grade).
tff(f_decl,type,f: grade).
tff(grade_of_decl,type,grade_of: human > grade).
tff(created_equal_decl,type,created_equal: (human * human) > $o).
tff(esk1_1_decl,type,esk1_1: human > human).

tff(con1,conjecture,
    ? [X4: human] : ( ( X4 != john ) & created_equal(X4,john) ),
    file('SomeoneNotJohn.p',there_is_someone_else) ).

tff(ax1,axiom,
    ! [X1: human,X2: human] : created_equal(X1,X2),
    file('SomeoneNotJohn.p',all_created_equal) ).

tff(ax2,axiom,
    grade_of(john) = f,
    file('SomeoneNotJohn.p',john_got_an_f) ).

tff(ax3,axiom,
    ! [X1: human] : ? [X2: human] :
      ( ( X1 != X2 ) & ( grade_of(X2) = a ) ),
    file('SomeoneNotJohn.p',someone_got_an_a) ).

tff(ax4,axiom,
    a != f,
    file('SomeoneNotJohn.p',f_is_not_a) ).

tff(inf1,negated_conjecture,
    ~ ? [X1: human] : ( X1 != john & created_equal(X1,john) ),
    inference(assume_negation,[status(cth)],[con1]) ).

tff(inf2,negated_conjecture,
    ! [X1: human] : ( X1 = john | ~ created_equal(X1,john) ),
    inference(split_conjunct,[status(thm)],[inf1]) ).

tff(inf3,plain,
    ! [X1: human] : X1 = john,
    inference(cn,[status(thm)],
      [inference(rw,[status(thm)],[inf2,ax1])]) ).

tff(inf4,plain,
    ! [X1: human] :
      ( X1 != esk1_1(X1) & grade_of(esk1_1(X1)) = a ),
    inference(skm,[status(esa),new_symbols(skolem,[esk1_1]),
                   skolemized(X2)],[ax3]) ).

tff(inf5,plain,
    ! [X1: human] : grade_of(esk1_1(X1)) != f,
    inference(rw,[status(thm)],
      [inference(split,[status(thm)],[inf4]),ax4]) ).

tff(proof,plain,
    $false,
    inference(sr,[status(thm)],
      [inference(rw,[status(thm)],[inf3,ax2]),inf5]) ).
%------------------------------------------------------------------
```

Figure 4: Example proof

isfiable. Unsatisfiable axioms are acceptable only in certain cases, e.g., the single parent is the negated conjecture, or the inferred formula is $false$, and a (W) or (E) is issued appropriately.

As always, calls to ATP systems are subject to resource constraints, and the systems might not produce results because a resource limit has been reached. Therefore "(E)"s in these cases are often not indications of errors, and should be examined manually.

**Structural verification** deals with non-logical aspects of a proof, checking whether the formulae presented as proof have the right format and relationships. The checks are:

1. Check the syntax of the annotated formulae. (E)
2. Check that the annotated formulae are uniquely named. (E)
3. Check that all the parents of inferred formulae exist. (E)
4. Check that all the annotated formulae in the output are actually used in the proof. (W)
5. Check that the derivation is acyclic. (E)
6. If the proof is expected to be a refutation, check that all roots are $false$ (there can be multiple $false$ when explicit splitting is used, as explained below), and that there is a negated conjecture. (E)
7. Check that all assumptions are propagated and discharged. (E)

**Leaf verification** deals with the leaves of the derivation, and their relationship with the problem formulae. The checks are:

1. Check-by-ATP that the leaf axioms are satisfiable. (W)
2. Check that introduced leaves are acceptable, e.g., definitions (Egly and Rath 1996; Reger, Suda, and Voronkov 2016), assumptions (Urban and Sutcliffe 2009), tautologies. (E)
3. Check that non-introduced leaves are copies of problem formulae, or can be proved from problem formulae using check-by-ATP. (E)

**Rule specific verification** deals with inference rules that require special treatment. The checks are:

1. Check explicit splitting (Weidenbach 2001) (E).
2. Check-by-ATP Skolemization. (E) Verification of Skolemization steps, including the techniques used in GDV-LP, is discussed in Section 2.3.
3. Check-by-ATP local steps of proof by contradiction. (E)

**Inference verification** deals with the various types of inferences that are made by ATP systems in a proof. The checks are:

1. For inference steps with SZS status `THM`, check-by-ATP that the inferred formula can be proved from the parent formulae. (E)
2. For inference steps with SZS status `CTH`, check-by-ATP that the negation of the inferred formula can be proved from the parent formulae. (E)
3. For inference steps with SZS status `ESA`, GDV attempts to prove equivalence, which is stronger than equisatifiability, and can be allowed to fail. This is implemented by a check-by-ATP that the inferred formula can be proved from the parent formulae, and a check-by-ATP that the parent formulae can be proved from the inferred formula. (W)

   A particular `ESA` case is Skolemization, which requires rule specific verification as explained above.
4. For inference steps with SZS status `ECS`, GDV attempts to prove counterequivalence, which is stronger than equicountersatifiability, and can be allowed to fail. This

is implemented by a check-by-ATP that the negation of the inferred formula can be proved from the parent formulae, and a check-by-ATP that the parent formulae can be proved from the negation of the inferred formula. (W)

Figure 5 shows the problem and proof DAG from Figures 3 and 4, with some of the checks listed above:

- Reading in the problem and proof formulae implements structural check 1.
- Traversing the formulae and links implements structural checks 2-6.
- Passing {ax1,ax2,ax3,ax4} to a trusted model finder is an example of leaf check 1.
- The four `Copied` links are examples of structural check 3 for the "copied" case.
- Passing {f_is_not_a} $\models$ ax4 to a trusted theorem prover is an example of leaf check 3 for the "inferred" case.
- Passing {ASk(ax3)} $\models$ inf4 to a trusted theorem prover is an example of rule specific check 2.
- Passing {ax2,inf3,inf5} $\models$ proof to a trusted theorem prover is an example of inference check 1.
- Passing {con1} $\models$ ~inf1 to a trusted theorem prover is an example of inference check 2.

The default trusted theorem provers and model finders used by GDV are Otter (McCune 2003) and E (Schulz, Cruanes, and Vukmirović 2019) for theorem proving, Paradox (Claessen and Smallbone 2018), Vampire (Kovacs and Voronkov 2013), and Nitpick (Blanchette and Nipkow 2010) for model finding. These systems have become trusted through the process described in Section 1.1. The level of trust varies, from very high in those systems that have been stable and empirically verified for many years, e.g., Otter, and Paradox, to systems that have been developed more recently but have undergone extensive testing, e.g., E and Vampire. The choice of trusted ATP systems can be changed through command line parameters to GDV, to satisfy the user's trust in particular systems.

### 2.3 Verifying Skolemization Steps

GDV currently has two approaches to verification of Skolemization steps. The older approach is to try to prove equivalence between the parent formula and its inferred Skolemized form – inference check 3.. As might be expected, the check that the Skolemized formula can be inferred from the parent normally fails, and a warning is issued.

The present pragmatic approach is to use a trusted Skolemizer to produce a trusted Skolemization of the parent formula, and then use a trusted theorem prover to check-by-ATP that the inferred formula can be proved from the trusted Skolemization. This relies on the trusted Skolemizer using the same Skolem symbol as in the proof, and Skolemizing the same variable, which in turn relies on the ATP system providing that information in the `inference` of the inferred annotated formula. If the ATP system does not provide the information then GDV falls back on the older approach. The trusted Skolemizer is ASk (Steen 2024), which takes the Skolem symbol and variable as parameters. As Skolemization steps are `esa` - EquiSatisfiable steps, GDV also does a check-by-ATP that the parent formula can be proved from the inferred formula.

In the future a third option will be implemented, appealing to higher-order techniques. A Hilbert $\epsilon$-term will be produced to define each Skolem term. Leaving the production of the $\epsilon$-term to the ATP system that produced the proof would leave an opening for errors in the proof, so a trusted $\epsilon$-term producer will be used. This approach will allow the Skolemization step to be checked by LambdaPi (see Section 3.1).

## 3 GDV + LambdaPi = GDV-LP

### 3.1 Dedukti and LambdaPi

Dedukti (Blanqui et al. 2023) is a formal language for defining theories in the $\lambda\Pi$-calculus modulo rewriting. It is an extension of Edinburgh's Logical Framework LF (Harper, Honsell, and Plotkin 1993) where types are identified not only modulo $\beta$-equivalence but also modulo user-defined rewriting rules. It allows for the representation of proofs of many different logical systems, from first-order to higher-order logic, the calculus of constructions, and some extensions of it (Blanqui et al. 2023), using the Curry-de Bruijn-Howard isomorphism between propositions and types, and proofs and $\lambda$-terms.

LambdaPi (Hondet and Blanqui 2020) is a proof assistant for the $\lambda\Pi$-calculus modulo rewriting, and can read and output Dedukti files. LambdaPi has a syntax that is more user-friendly than the Dedukti syntax, and provides some features that are useful when a direct translation of proofs to Dedukti is too difficult because the source proof is missing some information that Dedukti expects, e.g., LambdaPi supports implicit arguments/coercions, unification hints, proof tactics, etc., which are useful when representing, e.g., PVS terms (**?**), Alethe proofs from SMT solvers (**?**).

Tools have been developed to export the proofs of many different systems to the Dedukti or LambdaPi languages (Coq, OpenTheory, HOL-Light, Isabelle, Lean, etc.). There exist various checkers for Dedukti files, e.g `lambdapi` (Hondet and Blanqui 2020), `dkcheck` (Saillard 2015), and kontroli (Färber 2022).

### 3.2 GDV-LP

GDV uses trusted ATP systems to produce proofs and models that serve as certificates for the steps in the proof being verified. Checking the trusted proofs and models with an independent system provides another layer of certification and assurance that the original proof being verified is logically correct. GDV-LP provides this assurance layer using LambdaPi. To this end, ZenonModulo (Delahaye et al. 2013) is used for all steps where a check-by-ATP proof is required.[2] ZenonModulo is configured to output a LambdaPi term for each proof.

In order to use ZenonModulo's LambdaPi terms, GDV-LP produces the necessary files that declare the formulae, the signatures of the symbols in the terms, a LambdaPi term

[2]ZenonModulo can deal with only CNF, FOF, and TFF proofs. In the future Leo-III (Steen and Benzmüller 2018), and possibly also Vampire, will be able to produce LambdaPi terms, thus extending GDV-LP to TXF and THF proofs.

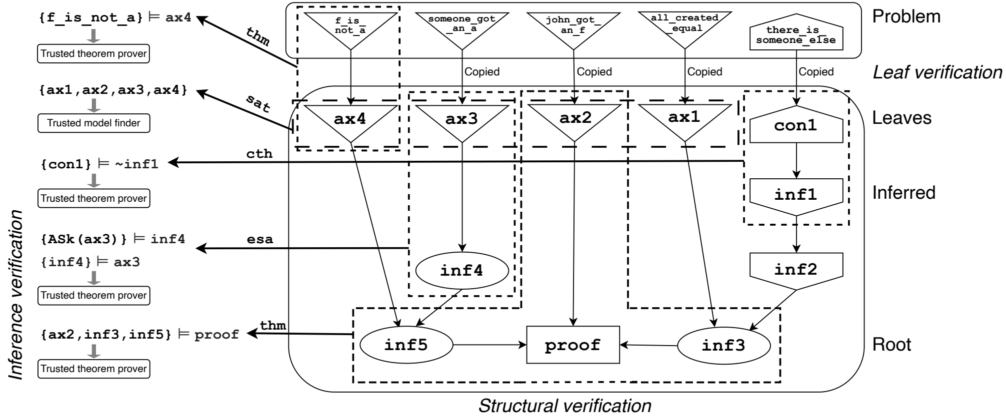


Figure 5: GDV architecture

for the root of the proof, and a lambdapi package. ZenonModulo's LambdaPi terms are chained together from the root term, and passed to lambdapi to be checked. (This approach was also taken in the EKSTRAKTO (Yacine El Haddad, Burel, and Blanqui 2019) verifier, but was limited to CNF refutations.) A strength of this added layer is that tools other than lambdapi can also be used to check the LambdaPi terms, e.g., dkcheck and kontroli.

GDV-LP has been tested on proofs from the TSTP solution library (Sutcliffe 2010), with a 30s CPU time limit on each ATP system run. The initial testing was on the 255 proofs found by E 3.2.0, for the 287 first-order theorems in the SYN domain (i.e., E did not find a proof for 32 problems). E was chosen because it reliably outputs well formed and structurally correct proofs. The proofs have a range of syntactic characteristics: from 5 to 843 formulae in the proof, from 4 to 797 inference steps, and a proof depth from 4 to 101. A time limit of 30s was imposed on each proof attempt by ZenonModulo, and also on LambdaPi for verifying the chained LambdaPi terms. GDV-LP verified 193 of the 255 proofs (75%). Of the 62 not verified, 47 (75%) were due to ZenonModulo not being able to find a proof in 30s, and 5 due to LambdaPi not being able to verify the chained LambdaPi term in 30s. Over the 193 verified proofs, there were 2211 inference steps verified by ZenonModulo for their SZS THM status. E 3.2.0 does not output the necessary information for Skolemization steps to be verified using trusted Skolemizations from ASk, and thus GDV-LP fell back on the older equivalence checking. There were 166 instances of this over 156 verified proofs, of which 18 instances were incomplete (the remaining 148 instances were complete because the parent and inferred formulae were unsatisfiable so that the verification of the steps used proof-by-contradiction).

GDV-LP is implemented in C, and uses the SystemOnTPTP framework (Sutcliffe 2000) to execute the trusted ATP systems and tools. GDV-LP is available from github.com/TPTPWorld/GDV.git.

# 4 Conclusion

This paper has described a framework by which ATP systems can become trusted. The sequence of finding a proof, verifying the proof, and certifying the verification, builds an increasing level of trust in the ATP system. This paper has traced one such path for TPTP format proofs generated by ATP systems, via the GDV derivation verifier, and ending at the LambdaPi checker. The verification steps have been implemented in the GDV-LP tool.

"Trust" is a key notion in (this form of) proof verification. There are multiple instances of trust in GDV-LP:

- External sources, including other ATP systems, are trusted to have produced correct expectations against which results can be checked.
- Parsers are trusted to check that proofs are syntactically well-formed.
- The GDV core is trusted to check the structure of proofs.
- ASk is trusted to perform correct Skolemizations.
- Various ATP systems are trusted for inference verification: model finders are trusted to check proofs' leaves for satisfiability, and theorem provers are trusted to check various inferences.
- ZenonModulo is trusted to produce LambdaPi terms that verify inference steps.
- GDV-LP is trusted to chain LambdaPi terms together.
- lambdapi is trusted to check a chain of LambdaPi terms.

Future work includes using LambdaPi to check Skolemization steps, verification of global inferences such as Vampire's (Kovacs and Voronkov 2013) consistent_polarity_flipping, and use of Leo-III or Vampire as the trusted theorem prover producing LambdaPi terms. The principles and implementation described in this paper are expected to be used in the proposed IJCAR 2026 Verifier & Verifiability Competition - the "ProoVer" competition. In the big picture, these notions of trust will naturally expand upstream, as trusted ATP systems are used to verify the output of subsymbolic reasoning systems, e.g., in AlphaProof (Hubert, Mehta, and Sartran 2024).

## References


[Al Wardani, Chaudhuri, and Miller 2023] Al Wardani, F.; Chaudhuri, K.; and Miller, D. 2023. Formal Reasoning Using Distributed Assertions. In Sattler, U., and Suda, M., eds., *Proceedings of the 14th International Symposium on Frontiers of Combining Systems*, number 14279 in Lecture Notes in Computer Science, 176–194. Springer-Verlag.

[Andreotti, Lachnitt, and Barbosa 2023] Andreotti, B.; Lachnitt, H.; and Barbosa, H. 2023. Carcara: An Efficient Proof Checker and Elaborator for SMT Proofs in the Alethe Format. In Sankaranarayanan, S., and Sharygina, N., eds., *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 13993 in Lecture Notes in Computer Science, Online. Springer-Verlag.

[Blanchette and Nipkow 2010] Blanchette, J., and Nipkow, T. 2010. Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder. In Kaufmann, M., and Paulson, L., eds., *Proceedings of the 1st International Conference on Interactive Theorem Proving*, number 6172 in Lecture Notes in Computer Science, 131–146. Springer-Verlag.

[Blanqui et al. 2023] Blanqui, F.; Dowek, G.; Grienenberger, E.; Hondet, G.; and Thiré, F. 2023. A Modular Construction of Type Theories. *Logical Methods in Computer Science* 19(1).

[Claessen and Smallbone 2018] Claessen, K., and Smallbone, N. 2018. Efficient Encodings of First-Order Horn Formulas in Equational Logic. In Galmiche, D.; Schulz, S.; and Sebastiani, R., eds., *Proceedings of the 9th International Joint Conference on Automated Reasoning*, number 10900 in Lecture Notes in Computer Science, 388–404.

[Coltellacci, Dowek, and Merz 2024] Coltellacci, A.; Dowek, G.; and Merz, S. 2024. Reconstruction of smt proofs with lambdapi. In *Proceedings of the 22nd International Workshop on Satisfiability Modulo Theories, , CEUR Workshop Proceedings 3725*.

[Cook 2018] Cook, B. 2018. Formal Reasoning About the Security of Amazon Web Services. In Chockler, H., and Weissenbacher, G., eds., *Proceedings of the 30th International Conference on Computer Aided Verification*, number 10981 in Lecture Notes in Computer Science, 38–47. Springer-Verlag.

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

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

[Egly and Rath 1996] Egly, U., and Rath, T. 1996. On the Practical Value of Different Definitional Translations to Normal Form. In McRobbie, M., and Slaney, J., eds., *Proceedings of the 13th International Conference on Automated Deduction*, number 1104 in Lecture Notes in Artificial Intelligence, 403–417. Springer-Verlag.

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

[Gordon, Milner, and Wadsworth 1979] Gordon, M.; Milner, A.; and Wadsworth, C. 1979. *Edinburgh LCF: A Mechanized Logic of Computation*. Number 78 in Lecture Notes in Computer Science. Springer.

[Hähnle and Huisman 2019] Hähnle, R., and Huisman, M. 2019. Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools. In Steffen, B., and Woeginger, G., eds., *Computing and Software Science: State of the Art and Perspectives*, number 10000 in Lecture Notes in Computer Science. Springer-Verlag. 345–373.

[Harper, Honsell, and Plotkin 1993] Harper, R.; Honsell, F.; and Plotkin, G. 1993. A Framework for Defining Logics. *Journal of the ACM* 40(1):143–184.

[Hondet and Blanqui 2020] Hondet, G., and Blanqui, F. 2020. The New Rewriting Engine of Dedukti. In Ariola, Z., ed., *Proceedings of the 5th International Conference on Formal Structures for Computation and Deduction*, number 167 in Leibniz International Proceedings in Informatics, 35:1–35:16. Dagstuhl Publishing.

[Hondet and Blanqui 2021] Hondet, G., and Blanqui, F. 2021. Encoding of predicate subtyping and proof irrelevance in the $\lambda\pi$-calculus modulo theory. In *Proceedings of the 26th International Conference on Types for Proofs and Programs, Leibniz International Proceedings in Informatics 188*.

[Hubert, Mehta, and Sartran 2024] Hubert, T.; Mehta, R.; and Sartran, L. 2024. AI Achieves Silver-medal Standard Solving International Mathematical Olympiad Problems. https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level/.

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

[McCune 2003] McCune, W. 2003. Otter 3.3 Reference Manual. Technical Report ANL/MSC-TM-263, Argonne National Laboratory, Argonne, USA.

[Reger, Suda, and Voronkov 2016] Reger, G.; Suda, M.; and Voronkov, A. 2016. New Techniques in Clausal Form Generation. In Benzmüller, C.; Sutcliffe, G.; and Rojas, R., eds., *Proceedings of the 2nd Global Conference on Artificial Intelligence*, number 41 in EPiC Series in Computing, 11–23. EasyChair Publications.

[Reger 2016] Reger, G. 2016. Better Proof Output for Vampire. In Kovacs, L., and Voronkov, A., eds., *Proceedings of the 3rd Vampire Workshop*, number 44 in EPiC Series in Computing, 46–60. EasyChair.

[Robinson and Voronkov 2001] Robinson, A., and Voronkov, A. 2001. *Handbook of Automated Reasoning*. Elsevier Science.

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

[Schlichtkrull et al. 2020] Schlichtkrull, A.; Blanchette, J.; Traytel, D.; and Waldmann, U. 2020. Formalizing Bachmair and Ganzinger's Ordered Resolution Prover. *Journal of Automated Reasoning* 64(7):1169–1195.

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

[Schulz 2006] Schulz, S. 2006. Algorithms and Data Structures for First-Order Equational Deduction. In Benzmüller, C.; Fischer, B.; and Sutcliffe, G., eds., *Proceedings of the 6th International Workshop on the Implementation of Logics*, number 212 in CEUR Workshop Proceedings, 1–6.

[Steen and Benzmüller 2018] Steen, A., and Benzmüller, C. 2018. The Higher-Order Prover Leo-III. In Galmiche, D.; Schulz, S.; and Sebastiani, R., eds., *Proceedings of the 8th International Joint Conference on Automated Reasoning*, number 10900 in Lecture Notes in Artificial Intelligence, 108–116.

[Steen 2024] Steen, A. 2024. ask v0.2.2. DOI: 10.5281/zenodo.14181705.

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

[Sutcliffe, Denney, and Fischer 2005] Sutcliffe, G.; Denney, E.; and Fischer, B. 2005. Practical Proof Checking for Program Certification. In Sutcliffe, G.; Fischer, B.; and Schulz, S., eds., *Proceedings of the Workshop on Empirically Successful Classical Automated Reasoning*.

[Sutcliffe 2000] Sutcliffe, G. 2000. SystemOnTPTP. In McAllester, D., ed., *Proceedings of the 17th International Conference on Automated Deduction*, number 1831 in Lecture Notes in Artificial Intelligence, 406–410. Springer-Verlag.

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

[Sutcliffe 2008] Sutcliffe, G. 2008. The SZS Ontologies for Automated Reasoning Software. In Sutcliffe, G.; Rudnicki, P.; Schmidt, R.; Konev, B.; and Schulz, S., eds., *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, 38–49.

[Sutcliffe 2010] Sutcliffe, G. 2010. The TPTP World - Infrastructure for Automated Reasoning. In Clarke, E., and Voronkov, A., eds., *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, number 6355 in Lecture Notes in Artificial Intelligence, 1–12. Springer-Verlag.

[Sutcliffe 2023] Sutcliffe, G. 2023. The Logic Languages of the TPTP World. *Logic Journal of the IGPL* 31(6):1153–1169.

[Urban and Sutcliffe 2009] Urban, J., and Sutcliffe, G. 2009. ATP-based Cross Verification of Mizar Proofs: Method, Systems, and First Experiments. *Journal of Mathematics in Computer Science* 2(2):231–251.

[Van Gelder and Sutcliffe 2006] Van Gelder, A., and Sutcliffe, G. 2006. Extending the TPTP Language to Higher-Order Logic with Automated Parser Generation. In Furbach, U., and Shankar, N., eds., *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, number 4130 in Lecture Notes in Artificial Intelligence, 156–161. Springer-Verlag.

[Weidenbach 2001] Weidenbach, C. 2001. Combining Superposition, Sorts and Splitting. In Robinson, A., and Voronkov, A., eds., *Handbook of Automated Reasoning*. Elsevier Science. 1965–2011.

[Yacine El Haddad, Burel, and Blanqui 2019] Yacine El Haddad, M.; Burel, G.; and Blanqui, F. 2019. EKSTRAKTO A tool to reconstruct Dedukti proofs from TSTP files. In Reis, G., and Barbosa, H., eds., *Proceedings of the 6th Workshop on Proof eXchange for Theorem Proving*, number 399 in Electronic Proceedings in Theoretical Computer Science, 27–35.