A Formally Verified Checker for First-Order Proofs

Seulkee Baek ✉

Department of Philosophy, Carnegie Mellon University, United States

Abstract

The Verified TESC Verifier (VTV) is a formally verified checker for the new Theory-Extensible Sequent Calculus (TESC) proof format for first-order ATPs. VTV accepts a TPTP problem and a TESC proof as input, and uses the latter to verify the unsatisfiability of the former. VTV is written in Agda, and the soundness of its proof-checking kernel is verified in respect to a first-order semantics formalized in Agda. VTV shows robust performance in a comprehensive test using all eligible problems from the TPTP problem library, successfully verifying all but the largest 5 of 12296 proofs, with >97% of the proofs verified in less than 1 second.

2012 ACM Subject Classification Computing methodologies → Theorem proving algorithms

Keywords and phrases TESC, TPTP, TSTP, ATP

Digital Object Identifier 10.4230/LIPIcs...

Acknowledgements This work has been partially supported by AFOSR grant FA9550-18-1-0120. Thanks to Jeremy Avigad for a careful proofreading of drafts and advice for improvements.

1 Introduction

Modern automated reasoning tools are highly complex pieces of software, and it is generally no simple matter to establish their correctness. Bugs have been discovered in them in the past [18, 11], and more are presumably hidden in systems used today. One popular strategy for coping with the possibility of errors in automated reasoning is the *De Bruijn* criterion [2], which states that automated reasoning software should produce 'proof objects' which can be independently verified by simple checkers that users can easily understand and trust. In addition to reducing the trust base of theorem provers, the De Bruijn criterion comes with the additional benefit that the small trusted core is often simple enough to be formally verified itself. Such thoroughgoing verification is far from universal, but there has been notable progress toward this goal in various subfields of automated reasoning, including interactive theorem provers, SAT solvers, and SMT solvers.

One area in which similar developments have been conspicuously absent is first-order automated theorem provers (ATPs), where the lack of a machine-checkable proof format [17] precluded the use of simple independent verifiers. The Theory-Extensible Sequent Calculus (TESC) [1] is a new proof format for first-order ATPs designed to fill this gap. In particular, the format's small set of fined-grained inference rules makes it relatively easy to implement and verify its proof checker.

This paper presents the Verified TESC Verifier (VTV), a proof checker for the TESC format written and verified in Agda [3]. The aim of the paper is twofold. Its immediate purpose is to demonstrate the reliability of TESC proofs by showing precisely what is established by their successful verification using VTV. Many of the implementation issues we shall discuss, however, are relevant to any formalization of first-order logic and not limited

[1] TESC is a new proof format developed by the author, and there is a separate paper awaiting submission to other venues that provides details of its specification and comprehensive test results using the TPTP problem library. For more details, please refer to `https://github.com/skbaek/tesc/tree/itp` for a draft of the format paper, as well as the source code of the TESC toolchain.

to either VTV or Agda. Therefore, the techniques VTV uses to solve them may be useful for other projects that reason about and perform computations with first-order logic data structures.

Speaking of applicability to other systems, one may also ask: why Agda? This choice was motivated more by subjective preference and convenience (especially the ease of code extraction) than any meticulous comparison of the strengths of its alternatives. No serious effort was made to replace it with other systems (e.g. Coq or Isabelle/HOL), but it is entirely possible that they would have been just as suitable.

The rest of the paper is organized as follows: Section 2 gives a brief survey of similar works and how VTV relates to them. Section 3 describes the syntax and inference rules of the TESC proof calculus. Section 4 presents the main TESC verifier kernel, and Section 5 gives a detailed specification of the verifier's soundness property. Sections 3, 4, and 5 also include code excerpts and discuss how their respective contents are formalized in Agda. Section 6 shows the results of empirical tests measuring VTV's performance. Section 7 gives a summary and touches on potential future work.

2 Related Works

SAT solving is arguably the most developed subfield of automated reasoning in terms of verified proof checkers. A non-exhaustive list of SAT proof formats with verified checkers include RAT [13], RUP and IORUP [14], LRAT [6], and GRIT [7]. In the related field of SMT solving, the SMTCoq project [1] also uses a proof checker implemented and verified in the Coq proof assistant.

Despite the limitations imposed by Gödel's second incompleteness theorem [10], there has been interesting work toward verification of interactive theorem provers. All of HOL Light except the axiom of infinity has been proven consistent in HOL Light itself [11], which was further extended later to include definitional principles for new types and constants [15]. There are also recent projects that go beyond the operational semantics of programming languages and verifies interactive theorem provers closer to the hardware, such as the Milawa theorem prover which runs on a Lisp runtime verified against x86 machine code [8], and the Metamath Zero [4] theorem prover which targets x84-64 instruction set architecture.

VTV is designed to serve a role similar to these verified checkers for first-order ATPs and the TESC format. There has been several different approaches [22, 5] to verifying the output of first-order ATPs, but the only example the author is aware of which uses independent proof objects with a verified checker is the Ivy system [16]. The main difference between Ivy and VTV is that Ivy's proof objects only record resolution and paramodulation steps, so all input problems have to be normalized by a separate preprocessor written in ACL2. In contrast, the TESC format supports preprocessing steps like Skolemization and CNF normalization, and allows ATPs to work directly on input problems with their optimized preprocessing strategies.

3 Proof Calculus

The syntax of the TESC calculus is as follows:

$$
\begin{aligned}
f &::= \sigma \mid \#_k \\
t &::= x_k \mid f(\vec{t}) \\
\vec{t} &::= \cdot \mid \vec{t}, t \\
\phi &::= \top \mid \bot \mid f(\vec{t}) \mid \neg\phi \mid \phi \lor \chi \mid \phi \land \chi \mid \phi \to \chi \mid \phi \leftrightarrow \chi \mid \forall\phi \mid \exists\phi \\
\Gamma &::= \cdot \mid \Gamma, \phi
\end{aligned}
$$

We let f range over *functors*, which are usually called 'non-logical symbols' in other presentations of first-order logic. The TESC calculus makes no distinction between function and relation symbols, and relies on the context to determine whether a symbol applied to arguments is a term or an atomic formula. For brevity, we borrow the umbrella term 'functor' from the TPTP syntax and use it to refer to any non-logical symbol.

There are two types of functors: σ ranges over *plain functors*, which you can think of as the usual relation or function symbols. We assume that there is a suitable set of symbols Σ, and let $\sigma \in \Sigma$. Symbols of the form $\#_k$ are *indexed functors*, and the number k is called the *functor index* of $\#_k$. Indexed functors are used to reduce the cost of introducing fresh functors: if you keep track of the largest functor index k that occurs in the environment, you may safely use $\#_{k+1}$ as a fresh functor without costly searches over a large number of terms and formulas. [2] This is similar to the indexing tricks used in other systems (e.g. HOL theorem provers) which use the maximum variable indices of theorems to make sure that two theorems have disjoint sets of variables.

We use t to range over terms, $\vec{t}$ over lists of terms, ϕ over formulas, and Γ over sequents. Quantified formulas are written without variables thanks to the use of De Bruijn indices [9]; the number k in variable x_k is its De Bruijn index. As usual, parentheses may be inserted for scope disambiguation, and the empty list operator '$\cdot$' may be omitted when it appears as part of a complex expression. E.g., the sequent $\cdot, \phi, \psi$ and term $f(\cdot)$ may be abbreviated to ϕ, ψ and f.

Formalization of TESC syntax in Agda is mostly straightforward, but with one small caveat. Consider the following definition of the type of terms:

```
data Term : Set where
  var : Nat → Term
  fun : Functor → List Term → Term
```

The constructor fun builds a complex term out of a functor and a list of arguments. Since these arguments are behind a List, they are not immediate subterms of the complex term, and Agda cannot automatically prove termination for recursive calls that use them. For instance, term-vars<? : Nat → Term → Bool is a function such that term-vars<? k t evaluates to true iff all variables in t have indices smaller than k. It would be natural to define this function as

```
term-vars<? : Nat → Term → Bool
term-vars<? k (var m) = m <ᵇ k
term-vars<? k (fun _ ts) = all (term-vars<? k) ts
```

[2] Thanks to Marijn Heule for suggesting this idea.

Rule	**Conditions**	**Example**
$\dfrac{\Gamma, A(b, \Gamma[i])}{\Gamma}\, A$		$\dfrac{\phi \leftrightarrow \psi, \phi \rightarrow \psi}{\phi \leftrightarrow \psi}\, A$
$\dfrac{\Gamma, B(0, \Gamma[i]) \qquad \Gamma, B(1, \Gamma[i])}{\Gamma}\, B$		$\dfrac{\phi \vee \psi, \phi \qquad \phi \vee \psi, \psi}{\phi \vee \psi}\, B$
$\dfrac{\Gamma, C(t, \Gamma[i])}{\Gamma}\, C$	$\text{lfi}(t) \leq \text{size}(\Gamma)$	$\dfrac{\neg\exists f(x_0), \neg f(g)}{\neg\exists f(x_0)}\, C$
$\dfrac{\Gamma,\ D(\text{size}(\Gamma), \Gamma[i])}{\Gamma}\, D$		$\dfrac{\exists f(x_0),\ f(\#_1)}{\exists f(x_0)}\, D$
$\dfrac{\Gamma, N(\Gamma[i])}{\Gamma}\, N$		$\dfrac{\neg\neg\phi, \phi}{\neg\neg\phi}\, N$
$\dfrac{\Gamma, \neg\phi \qquad \Gamma, \phi}{\Gamma}\, S$	$\text{lfi}(\phi) \leq \text{size}(\Gamma)$	$\dfrac{\neg f(\#_0) \qquad f(\#_0)}{\cdot}\, S$
$\dfrac{\Gamma, \phi}{\Gamma}\, T$	$\text{lfi}(\phi) \leq \text{size}(\Gamma)$, $\text{adm}(\text{size}(\Gamma), \phi)$	$\dfrac{= (f, f)}{\cdot}\, T$
$\dfrac{}{\Gamma}\, X$	For some i and j, $\Gamma[i] = \neg\Gamma[j]$	$\dfrac{}{\neg\phi, \phi}\, X$

Table 1 TESC inference rules. $\text{size}(\Gamma)$ is the number of formulas in Γ, and $\Gamma[i]$ denotes the (0-based) ith formula of sequent Γ, where $\Gamma[i] = \top$ if the index i is out-of-bounds. $\text{lfi}(x)$ is the largest functor index (lfi) occurring in x. If x incudes no functor indices, $\text{lfi}(x) = -1$. $\text{adm}(k, \phi)$ asserts that ϕ is an admissable formula in respect to the target theory and sequent size k.

where $m <^b k$ evaluates to true iff m is less than k. But Agda rejects this definition because it cannot prove that the recursive calls terminate. We can get around this problem by a mutual recursion between a pair of functions, one for terms and one for lists of terms:

```
term-vars<? : Nat → Term → Bool
terms-vars<? : Nat → List Term → Bool
term-vars<? k (var m) = m <ᵇ k
term-vars<? k (fun _ ts) = terms-vars<? k ts
terms-vars<? _ [] = true
terms-vars<? k (t :: ts) = term-vars<? k t ∧ terms-vars<? k ts
```

All other functions that recurse on terms are also defined using similar mutual recursion.

The inference rules of the TESC calculus are shown in Table 1. The TESC calculus is a one-sided first-order sequent calculus, so having a valid TESC proof of a sequent Γ shows that Γ is collectively unsatisfiable. The A,B,C,D, and N rules are the *analytic* rules. Analytic rules are similar to the usual one-sided sequent calculus rules, except that each analytic rule is overloaded to handle several connectives at once. For example, consider the formulas $\phi \wedge \psi$, $\neg(\phi \vee \psi)$, $\neg(\phi \rightarrow \psi)$, and $\phi \leftrightarrow \psi$. In usual sequent calculi, you would need a different rule for each of the connectives $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$ to break down these formulas. But all four formulas are "essentially conjunctive" in the sense that the latter three are equivalent to $\neg\phi \wedge \neg\psi$, $\phi \wedge \neg\psi$, and $(\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$. So it is more convenient to handle all four of them with a single rule that analyzes a formula into its left and right conjuncts, which is the analytic A rule. Similarly, the B, C, D rules are used to analyze essentially disjunctive, universal, existential formulas, and the N rule performs double-negation elimination. For a complete list of formula analysis functions that show how each analytic rule breaks down formulas, see Appendix A. The analytic rules are a slightly modified adaptation of Smullyan's *uniform notation* for analytic tableaux [21], which is where they get their names from.

All rules are designed to preserve the invariant $\mathrm{lfi}(\Gamma) < \mathrm{size}(\Gamma)$ for every sequent Γ. We say that a sequent Γ is *good* if it satisfies this invariant. It is important that all sequents are good, because this is what ensures that the indexed functor $\#_{\mathrm{size}(\Gamma)}$ is fresh in respect to a sequent Γ.

Of the three remaining rules, S is the usual cut rule, and X is the axiom or init rule. The T rule may be used to add *admissable* formulas. A formula ϕ is admissable in respect to a target theory T and sequent size k if it satisfies the following condition:

- For any good sequent Γ that is satisfiable modulo T and $\mathrm{size}(\Gamma) = k$, the sequent Γ, ϕ is also satisfiable modulo T.

More intuitively, the T rule allows you to add formulas that preserve satisfiability. Notice that the definition of admissability, and hence the definition of well-formed TESC proofs, depends on the implicit target theory. This is the 'theory-extensible' part of TESC. The current version of VTV verifies basic TESC proofs that target the theory of equality, so it allows T rules to introduce equality axioms, fresh relation symbol definitions, and choice axioms. But VTV can be easily modified in a modular way to target other theories as well.

TESC proofs are formalized in Agda as an inductive type Proof, which has a separate constructor for each TESC inference rule:

```
data Proof : Set where
  rule-a : Nat → Bool → Proof → Proof
  rule-b : Nat → Proof → Proof → Proof
  rule-c : Nat → Term → Proof → Proof
  rule-d : Nat → Proof → Proof
  rule-n : Nat → Proof → Proof
  rule-s : Formula → Proof → Proof → Proof
  rule-t : Formula → Proof → Proof
  rule-x : Nat → Nat → Proof
```

For example, the constructor rule-a takes Nat and Bool arguments because this is the information necessary to specify an application of the A rule. I.e., rule-a i b p is a proof whose last inference rule adds the formula $A(b, \Gamma[i])$. Notice that sequents are completely absent from the definition of Proof. This is a design choice made in favor of efficient space usage. Since TESC proofs are uniquely determined by their root sequents together with complete information of the inference rules used, TESC files conserve space by omitting any components that can be constructed on the fly during verification, which includes all intermediate sequents and formulas introduced by analytic rules. Proof is designed to only record information included in TESC files, since terms of the type Proof are constructed by parsing input TESC files.

4 The Verifier

The checker function verify for Proof performs two tasks: (1) it constructs the omitted intermediate sequents as it recurses down a Proof, and (2) it checks that the conditions are satisfied for each inference rule used. For instance, consider the definition of verify for the C rule case, together with its type signature:

```
verify : Sequent → Proof → Bool
verify Γ (rule-c i t p) =
  term-lfi<? (suc (size Γ)) t ∧
  verify (add Γ (analyze-c t (nth i Γ))) p
```

The terms (nth i Γ), (analyze-c t (nth i Γ)), and (add Γ (analyze-c t (nth i Γ))) each corresponds to $\Gamma[i]$, $C(t, \Gamma[i])$, and $\Gamma, C(t, \Gamma[i])$. The conjunct term-lfi<? (suc (size Γ)) t ensures the side condition $\mathrm{lfi}(t) \leq \mathrm{size}(\Gamma)$ is satisfied, and the recursive call to verify checks that the subproof p is a valid proof of the sequent $\Gamma, C(t, \Gamma[i])$. The cases for other rules are also defined similarly.

The argument type Sequent for verify presents some interesting design choices. What kind of data structures should be used to encode sequents? The first version of VTV used lists, but lists immediately become a bottleneck with practically-sized problems due to their poor random access speeds. The default TESC verifier included in the TPTP-TSTP-TESC Processor (T3P, the main tool for generating and verifying TESC files) uses arrays, but arrays are hard to come by and even more difficult to reason about in dependently typed languages like Agda. Self-balancing trees like AVL or red-black trees come somewhere between lists and arrays in terms of convenience and performance, but it can still be tedious to prove basic facts about them if those proofs are not available in your language of choice, as is the case for Agda's standard library.

For VTV, we cut corners by taking advantage of the fact that (1) formulas are never deleted from sequents, and (2) new formulas are always added to the right end of sequents. (1) and (2) allow us to use a simple balancing algorithm. In order to use the algorithm, we first define the type of trees in a way that allows each tree to efficiently keep track of its own size:

```
data Tree (A : Set) : Set where
  empty : Tree A
  fork : Nat → A → Tree A → Tree A → Tree A
```

For any non-leaf tree fork k a t s, the number k is the size of fork k a t s. This property is not guaranteed to hold by construction, but it is easy to ensure that it always holds in practice. Since sizes of trees are stored as Nat, the function size : Tree → Nat can always report the size of trees in constant time. New elements can be added to trees in a balanced way as follows: when adding an element to a non-leaf tree, compare its subtree sizes. If the right subtree is smaller, make a recursive call and add to the right subtree. If the sizes are equal, make a new tree that contains the current tree as its left subtree, an empty tree as its right subtree, and stores the new element in the root node. The definition of the add function for trees implements this algorithm precisely:

```
add : {A : Set} → Tree A → A → Tree A
add empty a = fork 1 a empty empty
add (fork k b t s) a =
  if (size s <ᵇ size t)
  then (fork (k + 1) b t (add s a))
  else (fork (k + 1) a (fork k b t s) empty)
```

This addition algorithm does not keep the tree maximally balanced, but it provides reasonable performance and does away with complex ordering and balancing mechanisms, which makes reasoning about trees considerably easier. Given the type Tree, the type Sequent can be simply defined as Tree Formula.

5 Soundness

In order to prove the soundness of verify, we first need to formalize a first-order semantics that it can be sound in respect to. Most of the formalization is routine, but it also includes some oddities particular to VTV.

One awkward issue that recurs in formalization of first-order semantics is the handling of arities. Given that each functor has a unique arity, what do you do with ill-formed terms and atomic formulas with the wrong number of arguments? You must either tweak the syntax definition to preclude such possibilities, or deal with ill-formed terms and formulas as edge cases, both of which can lead to bloated definitions and proofs.

For VTV, we avoid this issue by assuming that every functor has infinite arities. Or rather, for each functor f with arity k, there are an infinite number of other functors that share the name f and have arities $0, 1, ..., k-1, k+1, k+2, ...$ ad infinitum. With this assumption, the denotation of functors can be defined as

```
Rels : Set
Rels = List D → Bool

Funs : Set
Funs = List D → D
```

where D is the type of the domain of discourse that the soundness proof is parametrized over. A Rels (resp. Funs) can be thought of as a collection of an infinite number of relations (resp. functions), one for each arity. An interpretation is a pair of a relation assignment and a function assignment, which assign Rels and Funs to functors.

```
RA : Set
RA = Functor → Rels

FA : Set
FA = Functor → Funs
```

Variable assignments assign denotations to Nat, since variables are identified by their Bruijn indices.

```
VA : Set
VA = Nat → D
```

For reasons we've discussed in Section 3, term valuation requires a pair of mutually recursive functions in order to recurse on terms:

```
term-val : FA → VA → Term → D
terms-val : FA → VA → List Term → List D
term-val _ V (var k) = V k
term-val F V (fun f ts) = F f (terms-val F V ts)
terms-val F V [] = []
terms-val F V (t :: ts) = (term-val F V t) :: (terms-val F V ts)
```

Formula valuation is mostly straightforward, but some care needs to be taken in the handling of variable assignments and quantified formulas. The function qtf : Bool → Formula → Formula is a constructor of Formula for quantified formulas, where qtf false and qtf true encode the universal and existential quantifiers, respectively. Given a relation assignment R, function assignment F, and variable assignment V, the values of formulas qtf false ϕ and qtf true ϕ under R, F, and V are defined as

```
R , F , V ⊨ (qtf false ϕ) = ∀ x → (R , F , (V / 0 ↦ x) ⊨ ϕ)
R , F , V ⊨ (qtf true ϕ) = ∃ λ x → (R , F , (V / 0 ↦ x) ⊨ ϕ)
```

The notations $\forall\ x \rightarrow A$ and $\exists\ \lambda\ x \rightarrow A$ may seem strange, but they are just Agda's way of writing $\forall\ x\ A$ and $\exists\ x\ A$. For any V, k, and d, $(V\ /\ k \mapsto d)$ is an updated variable assignment obtained from V by assigning d to the variable x_k, and pushing the assignments of all variables larger than x_k by one. E.g., $(V\ /\ 0 \mapsto x)\ 0 = x$ and $(V\ /\ 0 \mapsto x)\ (\mathsf{suc}\ m) = V\ m$.

Now we can define (un)satisfiability of sequents in terms of formula valuations:

```
satisfies : RA → FA → VA → Sequent → Set
satisfies R F V Γ = ∀ ϕ → ϕ ∈ Γ → R , F , V ⊨ ϕ

sat : Sequent → Set
sat Γ = ∃ λ R → ∃ λ F → ∃ λ V → (respects-eq R × satisfies R F V Γ)

unsat : Sequent → Set
unsat Γ = ¬ (sat Γ)
```

Note that $\times$ is the constructor for product types, which serves the same role as a logical conjunction here. The respects-eq R clause asserts that the relation assignment R respects equality. This condition is necessary because we are targeting first-order logic with equality, and we are only interested in interpretations that satisfy all equality axioms.

VTV's formalization of first-order semantics is atypical in that (1) every functor doubles as both relation and function symbols with infinite arities, and (2) the definition of satisfiability involves variable assignments, thereby applying to open as well as closed formulas. These idiosyncracies, however, are harmless for our purposes: whenever a traditional interpretation (with unique arities for each functor and no variable assignment) M satisfies a set of sentences Γ, M can be easily extended to an interpretation in the above sense that still satisfies Γ, since the truths of sentences in Γ are unaffected by functors or variables that do not occur in them. Conversely, any satisfying interpretation in the above sense can be pruned into a traditinal one without changing truth values of sentences. Therefore, both definitions of satisfiability agree completely on sentences. Furthermore, we may assume that all formulas in an input sequent are sentences, because the TPTP parser will reject any open formula as it cannot assign a De Brujin index to an unbound variable. The impossibility of parsing an open formula, however, is property of a preprocessor stage written in Rust, and has not been verified.

Now we finally come to the soundness statement for verify:

```
verify-sound : ∀ (S : Sequent) (p : Proof) → good S → T (verify S p) → unsat S
```

T : Bool → Set maps Boolean values true and false to the trivial and empty sets $\top$ and $\bot$, respectively. The condition good S is necessary, because the soundness of TESC proofs is dependent on the invariant that all sequents are good (in the sense we defined in Section 3). But we can do better than merely assuming that the input sequent is good, because the parser which converts the input character list into the initial (i.e., root) sequent is designed to fail if the parsed sequent is not good. Therefore, we can prove this fact and use the success of the parser as confirmation that the parsed sequent is good. parse-verify is the outer function which accepts two character lists as argument, parses them into a Sequent and a Proof, and calls verify on them. The soundness statement for parse-verify is as follows:

$$\begin{array}{l} \textsf{parse-verify-sound} : \forall\ (\textit{seq-chars prf-chars} : \textsf{List Char}) \rightarrow \\ \quad \textsf{T}\ (\textsf{parse-verify}\ \textit{seq-chars prf-chars}) \rightarrow \\ \quad \exists\ \lambda\ (S : \textsf{Sequent}) \rightarrow \textsf{returns parse-sequent}\ \textit{seq-chars}\ S \times \textsf{unsat}\ S \end{array}$$

In other words, if parse-verify succeeds on two input character lists, then the sequent parser successfully parses the first character list into some unsatisfiable sequent S. parse-verify-sound is an improvement over verify-sound, but it also shows the limitation of the current setup. It only asserts that there is *some* unsatisfiable sequent parsed from the input characters, but provides no guarantees that this sequent is actually equivalent to the original TPTP problem. This means that the formal verification of VTV is limited to the soundness of its proof-checking kernel, and the correctness its TPTP parsing phase has to be taken on faith.

6 Test Results

The performance of VTV was tested by running it on all eligible problems in the TPTP [23] problem library. A TPTP problem is eligible if it satisfies all of the following conditions (parenthesized numbers indicate the total number of problems that satisfy all of the preceding conditions).

- It is in the CNF or FOF language (17053).
- Its status is 'theorem' or 'unsatisfiable' (13636).
- It conforms to the official TPTP syntax. More precisely, it does not have any occurrences of the character '%' in the `sq_char` syntactic class, as required by the TPTP syntax. This is important because T3P assumes that the input TPTP problem is syntactically correct and uses '%' as an endmarker (13389).
- All of its formulas have unique names. T3P requires this condition in order to unambiguously identify formulas by their names during proof compilation (13119).
- It can be solved by Vampire [19] or E [20] in one minute using default settings (Vampire = 7885, E = 4853 [3]).
- The TSTP solution produced by Vampire or E can be compiled to a TESC proof by T3P (Vampire = 7792, E = 4504).

The resulting $7792 + 4504 = 12296$ proofs were used for testing VTV. All tests were performed on Amazon EC2 `r5a.large` instances, running Ubuntu Server 20.04 LTS on 2.5 GHz AMD EPYC 7000 processors with 16 GB RAM. [4]

Out of the 12296 proofs, there were 5 proofs that VTV failed to verify due to exhausting the 16 GB available memory. A cactus plot of verification times for the remaining 12291 proofs are shown in Fig. 1. As a reference point, we also show the plot for the default TESC verifier included in T3P running on the same proofs. The default TESC verifier is written in

[3] It should be noted that the default setting is actually not optimal for E. When used with the `--auto` or `--auto-schedule` options, E's success rates are comparable to that of Vampire. But the TSTP solutions produced by E under these higher-performance modes are much more difficult to compile down to a TESC proof, so they could not be used for these tests.

[4] The aforementioned TESC repository (`https://github.com/skbaek/tesc/tree/itp`) also includes detailed information on the testing setup and a complete CSV of test results.

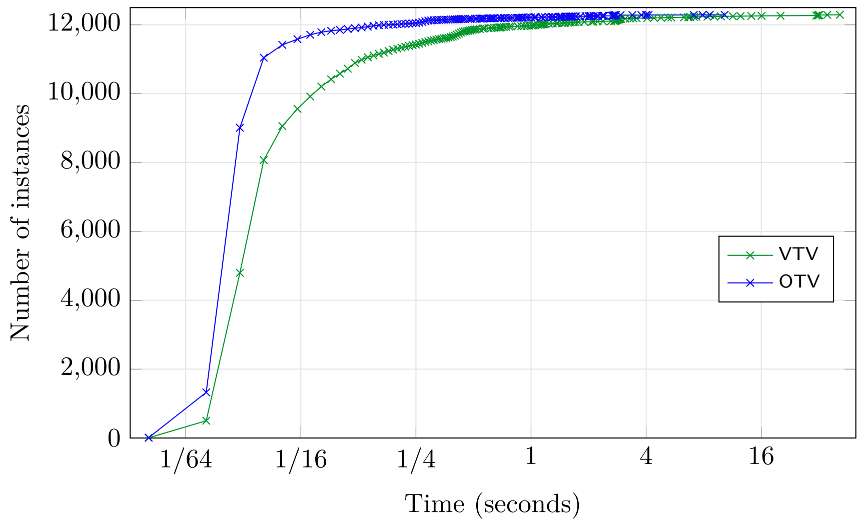

Figure 1 Verification times of VTV and OTV. The datapoints show the number of TESC proofs that each verifier could check within the given time limit. The plots look more "jumpy" toward the lower end due to the limited time measurement resolution (10 ms) of the unix `time` command.

Rust, and is optimized for performance with no regard to verification. For convenience, we refer to it as the Optimized TESC Verifier (OTV).

VTV is slower than OTV as expected, but the difference is unlikely to be noticed in actual use since the absolute times for most proofs are very short, and the total times are dominated by a few outliers. VTV verified >97.4% of proofs under 1 second, and >99.3% under 5 seconds. Also, the median time for VTV is 40 ms, a mere 10 ms behind the OTV's 30 ms. But OTV's mean time (54.54 ms) is still much shorter than that of VTV (218.93 ms), so users may prefer to use it for verifying one of the hard outliers or processing a large batch of proofs at once.

The main drawback of VTV is its high memory consumption. Fig. 2 shows the peak memory usages of the two verifiers. For a large majority of proofs, memory usages of both verifiers are stable and stay within the 14-20 MB range, but VTV's memory usage spikes earlier and higher than OTV. Due to the limit of the system used, memory usage could only be measured up to 16 GB, but the actual peak for VTV would be higher if we included the 5 failed verifications. A separate test running VTV on an EC2 instance with 64 GB ram (`r5a.2xlarge`) still failed for 3 of the 5 problematic proofs, so the memory requirement for verifying all 12296 proofs with VTV is at least >64 GB. In contrast, OTV could verify all 12296 proofs with less than 3.2 GB of memory.

It's not completely clear where the memory usage difference comes from, but it is most likely caused by the different ways in which VTV and OTV keep track of subgoal sequents. OTV uses a single main array to store the current goal sequent, and destructively updates this array whenever the goal sequent is changed. Consider, for instance, the application of the S rule to a sequent Γ, which creates subgoal sequents of the form $\Gamma, \neg\phi$ and Γ, ϕ. Before the rule application, OTV's main array holds Γ. Then it proceeds by:

1. Storing ϕ on a secondary array for later use
2. Pushing $\neg\phi$ onto the main array to change the goal sequent to $\Gamma, \neg\phi$
3. Verifying that the left subproof is a valid proof of the sequent $\Gamma, \neg\phi$
4. Truncating the main array back to Γ

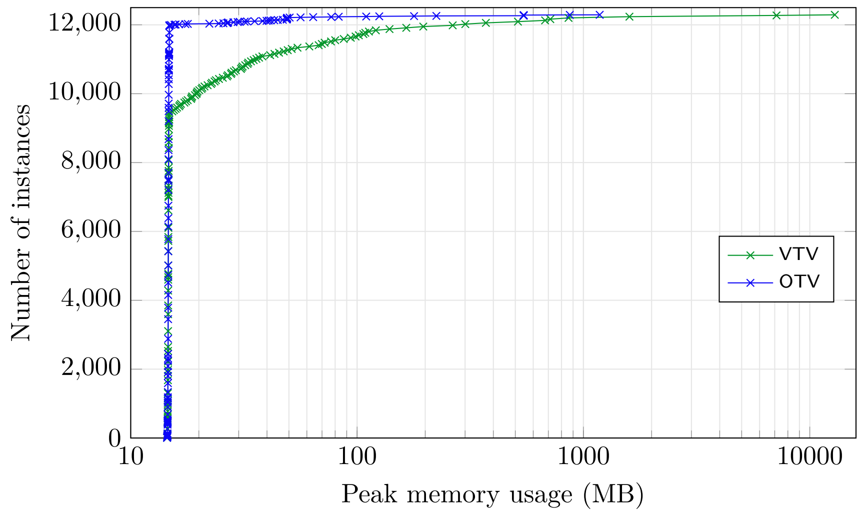

Figure 2 Peak memory usages of VTV and OTV. The datapoints show the number of TESC proofs that each verifier could check within the given peak memory usage.

5. Pushing ϕ onto the main array to change the goal sequent to Γ, ϕ
6. Verifying that the right subproof is a valid proof of the sequent Γ, ϕ

This means that OTV never allocates any memory to subgoal sequents that are already closed. On the other hand, VTV is an Agda program that gets compiled to Haskell, so it is almost certain that new memory will be allocated to both $\Gamma, \neg\phi$ and Γ, ϕ for this step. Furthermore, $\Gamma, \neg\phi$ will stay in memory even after it has been closed until the garbage collector deallocates it. So it is expected that VTV will use more memory than OTV, especially when verifying proofs that branch a lot and create large numbers of subgoals.

7 Conclusion

The robust test results show that VTV can serve as a fallback option when extra rigour is required in verification, thereby increasing our confidence in the correctness of TESC proofs. It can also help the design of other TESC verifiers by providing a reference implementation that is guaranteed to be correct. On a more general note, VTV is an example of a first-order logic formalization that strikes a practical balance between ease of proofs and efficient computation while avoiding some common pitfalls like non-termination and complex arity checking. Therefore, it can provide a useful starting point for other verified programming projects that use first-order logic.

There are two main ways in which VTV could be further improved. Curbing its memory usage would be the most important prerequisite for making it the default verifier in T3P. This may require porting VTV to a verified programming language with more efficient data structures (especially arrays) and finer-grained control over memory usage. The upcoming Lean 4 seems a promising candidate for this, considering its heavy emphasis on practical programming.

VTV could also benefit from a more reliable TPTP parser. A formally verified parser would be ideal, but the complexity of TPTP's syntax makes it difficult to even *specify* the correctness of a parser, let alone prove it. A more realistic approach would be imitating the technique used by verified LRAT checkers [12], making VTV print the parsed problem and textually comparing its output with the original problem. Due to TPTP's flexible syntax, however, the mapping from input files to printed outputs will necessarily be non-injective (e.g., there is no way to infer the positions of line breaks from a parsed formula). This means that the original problem will have to be normalized before comparison, and we must either verify or trust an additional TPTP normalizer.

References

1. Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A modular integration of sat/smt solvers to coq through proof witnesses. In *International Conference on Certified Programs and Proofs*, pages 135–150. Springer, 2011.
2. Henk Barendregt and Freek Wiedijk. The challenge of computer mathematics. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 363(1835):2351–2375, 2005.
3. Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.
4. Mario Carneiro. Metamath zero: The cartesian theorem prover. *arXiv preprint arXiv:1910.10703*, 2019.
5. Zakaria Chihani, Tomer Libal, and Giselle Reis. The proof certifier checkers. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 201–210. Springer, 2015.
6. Luís Cruz-Filipe, Marijn JH Heule, Warren A Hunt, Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified rat verification. In *International Conference on Automated Deduction*, pages 220–236. Springer, 2017.
7. Luís Cruz-Filipe, Joao Marques-Silva, and Peter Schneider-Kamp. Efficient certified resolution proof checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 118–135. Springer, 2017.
8. Jared Davis and Magnus O Myreen. The reflective milawa theorem prover is sound (down to the machine code that runs it). *Journal of Automated Reasoning*, 55(2):117–183, 2015.
9. Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. North-Holland, 1972.
10. Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für mathematik und physik*, 38(1):173–198, 1931.
11. John Harrison. Towards self-verification of hol light. In *International Joint Conference on Automated Reasoning*, pages 177–191. Springer, 2006.
12. Marijn Heule, Warren Hunt, Matt Kaufmann, and Nathan Wetzler. Efficient, verified checking of propositional proofs. In *International Conference on Interactive Theorem Proving*, pages 269–284. Springer, 2017.
13. Marijn JH Heule, Warren A Hunt, and Nathan Wetzler. Verifying refutations with extended resolution. In *International Conference on Automated Deduction*, pages 345–359. Springer, 2013.
14. Marijn JH Heule, Warren A Hunt Jr, and Nathan Wetzler. Bridging the gap between easy generation and efficient verification of unsatisfiability proofs. *Software Testing, Verification and Reliability*, 24(8):593–607, 2014.
15. Ramana Kumar, Rob Arthan, Magnus O Myreen, and Scott Owens. Hol with definitions: Semantics, soundness, and a verified implementation. In *International Conference on Interactive Theorem Proving*, pages 308–324. Springer, 2014.
16. William McCune and Olga Shumsky. Ivy: A preprocessor and proof checker for first-order logic. In *Computer-Aided reasoning*, pages 265–281. Springer, 2000.
17. Giles Reger and Martin Suda. Checkable proofs for first-order theorem proving. In *ARCADE@*

CADE, pages 55–63, 2017.
18 Giles Reger, Martin Suda, and Andrei Voronkov. Testing a saturation-based theorem prover: Experiences and challenges. In *International Conference on Tests and Proofs*, pages 152–161. Springer, 2017.
19 Alexandre Riazanov and Andrei Voronkov. The design and implementation of vampire. *AI communications*, 15(2, 3):91–110, 2002.
20 Stephan Schulz. E–a brainiac theorem prover. *Ai Communications*, 15(2, 3):111–126, 2002.
21 Raymond M Smullyan. *First-order logic*. Courier Corporation, 1995.
22 Geoff Sutcliffe. Semantic derivation verification: Techniques and implementation. *International Journal on Artificial Intelligence Tools*, 15(06):1053–1070, 2006.
23 Geoff Sutcliffe. The TPTP problem library and associated infrastructure. *Journal of Automated Reasoning*, 43(4):337, 2009.

A Formula analysis functions

A	$A(0, \neg(\phi \vee \psi)) = \neg\phi$	$A(1, \neg(\phi \vee \psi)) = \neg\psi$
	$A(0, \phi \wedge \psi) = \phi$	$A(1, \phi \wedge \psi) = \psi$
	$A(0, \neg(\phi \to \psi)) = \phi$	$A(1, \neg(\phi \to \psi)) = \neg\psi$
	$A(0, \phi \leftrightarrow \psi) = \phi \to \psi$	$A(1, \phi \leftrightarrow \psi) = \psi \to \phi$
B	$B(0, \phi \vee \psi) = \phi$	$B(1, \phi \vee \psi) = \psi$
	$B(0, \neg(\phi \wedge \psi)) = \neg\phi$	$B(1, \neg(\phi \wedge \psi)) = \neg\psi$
	$B(0, \phi \to \psi) = \neg\phi$	$B(1, \phi \to \psi) = \psi$
	$B(0, \neg(\phi \leftrightarrow \psi)) = \neg\phi \to \psi$	$B(1, \neg(\phi \leftrightarrow \psi)) = \neg\psi \to \phi$
C	$C(t, \forall\phi) = \phi[0 \mapsto t]$	$C(t, \neg\exists\phi) = \neg\phi[0 \mapsto t]$
D	$D(k, \exists\phi) = \phi[0 \mapsto \#_k])$	$D(k, \neg\forall\phi) = \neg\phi[0 \mapsto \#_k]$
N	$N(\neg\neg\phi) = \phi$	

The Formula analysis functions used by analytic rules. For any argument not explicitly defined above, the functions all return $\top$. E.g., $A(0, \phi \vee \psi) = \top$. $\phi[k \mapsto t]$ denotes the result of replacing all variables in ϕ bound to the kth quantifier with term t.