A Formally Verified TESC Verifier

Seulkee Baek

29th January, 2021

Abstract

This paper introduces the Verified TESC Verifier (VTV), 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 elligible problems from the TPTP problem library, successfully verifying all but the 3 largest proofs out of 12296 proofs, with >97% of the proofs verified under 1 second.

1 Introduction

Modern automated reasoning tools are highly complex software whose correctness cannot be easily established. Numerous bugs have been discovered in them in the past, and more are presumably hidden in sytems used today. One popular strategy for coping with the possibility of errors in automated reasoning is the *De Brujin* criterion [1], 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, adherence to the De Bruijn criterion comes with the additional benefit that the small trusted core is often simple enough to be formally verified themselves. Such thoroughgoing verification is far from universal, but there has been notable progress toward this goal in various subfields of automated reasonning, 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 precluded any simple independent verifiers, let alone a formally verified one. The Theory-Extensible Sequent Calculus (TESC) is a new proof format for first-order ATPs designed to address this gap. In particular, the format's small set of fined-grained inference rules makes it straightforward 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 []. The aim of the paper is twofold: primary aim = provide confidence in the TESC proof format. The statement and proof of VTV's soundness shows exactly what is entailed by a successful verification of a TESC proof. Hopes of increased developer interest in the new format Secondary aim = a guide for understanding the VTV codebase and possibly implementing your own TESC verifier, especially in dependently typed languages such as Lean or Coq.

The rest of the paper is organized as follows: Section 4 describes the syntax and inference rules of the TESC proof calculus. Section 5 presents the main TESC verifier, and Section 6 deals with the statement and proof of verifier correctness. Sections 4, 5, and 6 are also accompanied by code excerpts and discussions of the Agda formalization of their respective contents. Section 7 shows results of empirical tests that compare the performance of VTV against the default TESC verifier written in Rust. Section 2 briefly surveys related works. Section 8 gives a summary and discusses potential future work.

2 Related Works

SAT solving is arguably the most developed subfield of automated reasoning in terms of proof checker certification. A non-exhaustive list of verified verifiers for SAT proof formats include...

Despite the hard limits imposed by Goedel's second incompleteness theorem [], there has been interesting work toward self-verification of interactive theorem prover kernels. All of HOL Light except the axiom of infinity has been proven consistent in HOL Light itself [], which allows us to consider HOL Light consistent for most practical proofs (i.e., any proof that is not artificially contrived to exploit the missing axiom). More recent work on Metamath Zero [] aims to verify not only a large part of Metamath Zero's

logic, but also its implementation down to the level of x84-64 instruction set architecture.

The SMTCoq project [] also includes a checker for SMT proofs that has been formally verified in and extracted from Coq.

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 to verifying the output of ATPs, but (to the extent of my knowledge) none has featured a proof format with a verified checker. For instance, GDV [] works by breaking down an ATP-generated solution into small subproblems and re-solving them with multiple ATPs. The redundancy of multiple ATPs provides more reliability than any single solver, but there isn't much that can be verified about the operation of GDV short of verifying the highly complex ATPs used. Foundational Proof Certificates [] is a system that could potentially be used to specify proof formats and implement proof checkers for first-order ATPs, but its actual implementation has been limited to specification of the paramodulation rule and an unverified checker.

3 Conventions

'Its largest functor index is smaller than its size" is a mouthful, we simply say that a sequent Γ is *good* iff $\mathrm{lfi}(\Gamma) < \mathrm{size}(\Gamma)$.

Define 'functor'

4 Proof Calculus

As its name suggests, the TESC proof format is based on a first-order sequent calculus. Compared to typical first-order sequent calculi, the TESC calculus includes some oddities which facilitates mechanical proof construction and verification. The syntax of TESC terms and formulas are as follows:

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

t ranges over terms, $\vec{t}$ over lists of terms, and ϕ over formulas. Quantified formulas are written without variables due to the use of De Bruijn indices [];

the number k in variable x is its De Bruijn index. f and r are regular function and relation symbols. Symbols of the form $\#_k$ are *indexed functors*, a special syntactic class which may be used as either function or relation symbols. The number k of a indexed functor $\#_k$ is its *functor index*. Indexed functors are used to ensure that a newly introduced function or relation symbol is fresh: if you know that k is the largest functor index that occurs in the current environment, you may safely use $\#_{k+1}$ as a fresh symbol without having to search through the enviroment.

Formalization of TESC syntax in Agda is mostly straightforward, but with one caveat: the first-instinct definition of terms as

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

works poorly in practice, since any structural recursion on terms immediately runs into non-termination issues. We could try to manually prove termination, but it is much simpler to sidestep this issue with a pseudo-mutual recursion:

```
data Term* : Bool → Set where
  var : Nat → Term* false
  fun : Functor → Term* true → Term* false
  nil : Term* true
  cons : Term* false → Term* true → Term* true
```

which lets us define terms and lists of terms as

```
Term = Term* false
Terms = Term* true
```

The inference rules of the TESC calculus are shown in Table 1. The A,B,C,D, and N rules are *analytic* rules which analyze existing formulas on the sequent and adds resulting subformulas to the new sequent (we are reading the proof tree in the bottom-up direction). The formula analysis functions used in analytic rules are given in Table 2. Notice that the analytic rules are very similar to Smullyan's *uniform notation* for analytic tableaux, which is where they get their names from. $\Gamma[i]$ denotes the ith formula of sequent Γ, where $\Gamma[i] = \top$ if the index i is out-of-bounds. S is the usual

Rule	Conditions
$\frac{\Gamma, A(b, \Gamma[i])}{\Gamma} A$	
$\frac{\Gamma, B(0, \Gamma[i]) \qquad \Gamma, B(1, \Gamma[i])}{\Gamma} B$	
$\frac{\Gamma, C(t, \Gamma[i])}{\Gamma} C$	$\text{lfi}(t) \leq \text{size}(\Gamma)$
$\frac{\Gamma,\ D(\text{size}(\Gamma), \Gamma[i])}{\Gamma} D$	
$\frac{\Gamma, N(\Gamma[i])}{\Gamma} N$	
$\frac{\Gamma, \neg\phi \qquad \Gamma, \phi}{\Gamma} S$	$\text{lfi}(\phi) \leq \text{size}(\Gamma)$
$\frac{\Gamma, \phi}{\Gamma} T$	$\text{lfi}(\phi) \leq \text{size}(\Gamma),\ \text{adm}(\text{size}(\Gamma), \phi)$
$\frac{}{\Gamma} X$	$\exists i.\exists j.(\Gamma[i] = \neg\Gamma[j])$

Table 1: TESC inference rules.

cut rule, and X is the axiom or init rule. T is an extensible rule which may be used to add any formula that is admissible (hence the side condition $\text{adm}(k, \phi)$) in respect to the target theory. The current version of TESC only targets equality, so the T rule may be used to introduce equality axioms, fresh relation symbol definitions, and choice axioms. $\text{mfi}(x)$ denotes the largest functor index occuring in x. The mfi conditions for C, S, and T rules are necessary to ensure that the invariant $\text{mfi}(\Gamma) < \text{len}(\Gamma)$ is preserved for any sequent Γ, where $\text{len}(\Gamma)$ is the length of Γ.

TESC proofs are formalized in Agda as follows:

```
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
```

A	$A(0, \neg(\phi \vee \psi)) = \neg\phi$	$A(1, \neg(\phi \vee \psi)) = \neg\psi$
	$A(0, \phi \wedge \psi) = \phi$	$A(1, \phi \wedge \psi) = \psi$
	$A(0, \neg(\phi \to \psi)) = \phi$	$A(1, \neg(\phi \to \psi)) = \neg\psi$
	$A(0, \phi \leftrightarrow \psi) = \phi \to \psi$	$A(1, \phi \leftrightarrow \psi) = \psi \to \phi$
	For any other b and ϕ, $A(b, \phi) = \top$	
B	$B(0, \phi \vee \psi) = \phi$	$B(1, \phi \vee \psi) = \psi$
	$B(0, \neg(\phi \wedge \psi)) = \neg\phi$	$B(1, \neg(\phi \wedge \psi)) = \neg\psi$
	$B(0, \phi \to \psi) = \neg\phi$	$B(1, \phi \to \psi) = \psi$
	$B(0, \neg(\phi \leftrightarrow \psi)) = \neg\phi \to \psi$	$B(1, \neg(\phi \leftrightarrow \psi)) = \neg\psi \to \phi$
	For any other b and ϕ, $B(b, \phi) = \top$	
C	$C(t, \forall\phi) = \phi[0 \mapsto t]$	$C(t, \neg\exists\phi) = \neg\phi[0 \mapsto t]$
	For any other t and ϕ, $C(t, \phi) = \top$	
D	$D(k, \exists\phi) = \phi[0 \mapsto \#_k(\cdot)])$	$D(k, \neg\forall\phi) = \neg\phi[0 \mapsto \#_k(\cdot)]$
	For any other k and ϕ, $D(k, \phi) = \top$	
N	$N(\neg\neg\phi) = \phi$	
	For any other ϕ, $N(\phi) = \top$	

Table 2: Formula analysis functions.

```
rule-x : Nat → Nat → Proof
```

Note that there are parts of TESC proofs omitted in the definition of `Proof`, e.g. sequents. This is a design choice made in favor of efficient space usage. Since proofs are uniquely determined by their root sequents + complete information of the inference rules used, TESC proof files save 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. Terms of the type Proof are constructed by parsing input TESC files, so it only includes information stored in TESC files, which are the arguments to the constructors of `Proof`.

5 Verifier

Since `Proof` only includes basic information regarding inference rule applications, the verifier function for `Proof` must construct intermediate sequents as it recurses down a proof, and also check that inference rule arguments (e.g.,

the term t of a C-rule application) satisfy their side conditions. There are no surprises in the definition of `verify`:

```
verify : Sequent → Proof → Bool
verify Γ (rule-a i b p) = verify (add Γ (analyze-a b (nth i Γ))) p
verify Γ (rule-b i p q) =
  (verify (add Γ (analyze-b false (nth i Γ))) p) ∧
  (verify (add Γ (analyze-b true (nth i Γ))) q)
verify Γ (rule-c i t p) =
  term*-lfi<? (suc (size Γ)) t ∧
  verify (add Γ (analyze-c t (nth i Γ))) p
verify Γ (rule-d i p) = verify (add Γ (analyze-d (size Γ) (nth i Γ))) p
verify Γ (rule-n i p) = verify (add Γ (analyze-n (nth i Γ))) p
verify Γ (rule-s ϕ p q) =
  formula-lfi<? (suc (size Γ)) ϕ ∧
  verify (add Γ (not ϕ)) p ∧ verify (add Γ ϕ) q
verify Γ (rule-t ϕ p) =
  adm? (size Γ) ϕ ∧ verify (add Γ ϕ) p
verify Γ (rule-x i j) = formula=? (nth i Γ) (not (nth j Γ))
```

Analytic rules introduce new formulas obtained by formula analysis using `apply` functions, and side conditions are checked using appropriate `check` functions. The argument type `Sequent`, however, offers some interesting design choices. What kind of data structures should be used to encode sequents? The first version of VTV used lists, which is an intuitive choice based on the linear structure of sequents. But with practically-sized problems, lists immediately become a bottleneck due to their poor random access speeds. The default TESC verifier 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 be tedious to prove basic properties of such trees if those proofs are not available in your language of choice, which is also the case for Agda and its 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 end of sequents. This allows us to use a simple tree structure:

```
data Tree (A : Set) : Set where
```

```
empty : Tree A
fork : Nat → A → Tree A → Tree A → Tree A
```

For any tree `fork k a t s`, the number `k` is the size of the left subtree `t`. This property is not guaranteed to hold by construction, but it is easy to ensure that it always holds in practice. With this definition, balanced addition of elements to trees becomes trivial:

```
add : {A : Set} → Tree A → A → Tree A
add empty a = fork 1 a empty empty
add ts@(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 ts empty)
```

Then the type `Sequent` can be defined as `Tree Formula`.

6 Correctness

In order to verify the correctness of `verify`, we first need to formalize a a first-order semantics against which it can be correct. Although most of it is is routine, there are somem features particular to the formalization for 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 unpleasant bloat.

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 has arities $0, 1, ..., k-1, k+1, k+2, ...$ ad infinitum. With this assumption, the denotation of functors can be simply defined as

```
Rels : Set
Rels = List D → Bool

Funs : Set
Funs = List D → D
```

A `Rel` (resp. `Fun`) is a collection of an infinite number of relations (resp. functions), one for each arity. Interpretations are assignments of functors to `Rel` and `Fun`, plus assignments of variables to the domain of discourse. Note that `VA` may simply assign denotations to `Nat` due to the use of Bruijn indices.

```
RA : Set
RA = Functor → Rels

FA : Set
FA = Functor → Funs

VA : Set
VA = Nat → D
```

Now that we have first-order interpretations, we can define the value of terms and forms under an interpretation. Term valuation requires a bit of ingenuity due to the unusual definition of `Term*`:

```
ElemList : Set → Bool → Set
ElemList A false = A
ElemList A true = List A

term*-val : FA → VA → {b : Bool} → Term* b → ElemList D b
term*-val _ V (var k) = V k
term*-val F V (fun f ts) = F f (term*-val F V ts)
term*-val F V nil = []
term*-val F V (cons t ts) = (term*-val F V t) :: (term*-val F V ts)
```

Formula valuation recurses down the structure of `Formula`, and maps each logical connective to its equivalent in Agda's `Set`.

```
_,_,_⊨_ : RA → FA → VA → Formula → Set
R , F , V ⊨ (cst b) = ⊤ b
R , F , V ⊨ (not φ) = ¬ (R , F , V ⊨ φ)
R , F , V ⊨ (bct or φ ψ) = (R , F , V ⊨ φ) ⊎ (R , F , V ⊨ ψ)
R , F , V ⊨ (bct and φ ψ) = (R , F , V ⊨ φ) × (R , F , V ⊨ ψ)
R , F , V ⊨ (bct imp φ ψ) = (R , F , V ⊨ φ) → (R , F , V ⊨ ψ)
R , F , V ⊨ (bct iff φ ψ) = (R , F , V ⊨ φ) ↔ (R , F , V ⊨ ψ)
R , F , V ⊨ (qtf false φ) = ∀ x → (R , F , (V / 0 ↦ x) ⊨ φ)
```

$$R\ ,\ F\ ,\ V \vDash (\mathsf{qtf}\ \mathsf{true}\ \phi) = \exists\ \lambda\ x \to (R\ ,\ F\ ,\ (V\ /\ 0 \mapsto x) \vDash \phi)$$
$$R\ ,\ F\ ,\ V \vDash (\mathsf{rel}\ r\ ts) = \mathsf{T}\ (R\ r\ (\mathsf{term^*\text{-}val}\ F\ V\ ts))$$

`V/0↦x` is an variable assignment update which assigns a new denotation to the zeroth variable, and pushes all other assignments up by one. I.e., `(V/0↦x) 0 = x` and `(V/0↦x) (k+1) = V k` for all `k`. T is a function that maps true to $\top$ and false to $\bot$.

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

$$\mathsf{satisfies} : \mathsf{RA} \to \mathsf{FA} \to \mathsf{VA} \to \mathsf{Sequent} \to \mathsf{Set}$$
$$\mathsf{satisfies}\ R\ F\ V\ B = \forall\ f \to \mathsf{mem}\ f\ B \to R\ ,\ F\ ,\ V \vDash f$$

$$\mathsf{sat} : \mathsf{Sequent} \to \mathsf{Set}$$
$$\mathsf{sat}\ B = \exists\ \lambda\ R \to \exists\ \lambda\ F \to \exists\ \lambda\ V \to (\mathsf{respects\text{-}eq}\ R \times \mathsf{satisfies}\ R\ F\ V\ B)$$

$$\mathsf{unsat} : \mathsf{Sequent} \to \mathsf{Set}$$
$$\mathsf{unsat}\ B = \neg\ (\mathsf{sat}\ B)$$

The respects-eq R clause asserts that the relation assignment R respects equality. This condition is necessary because we are targetting first-order logic with equality; we are only interested in interpretations that satisfy all equality axioms.

Our formalization of first-order semantics is atypical in that (1) every non-logical symbol 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. But this is completely 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 not affected by functors or variables that do not occur in them. Therefore, if a set of sentences is unsatisfiable in the sense defined above, it is also unsatisfiable in the usual sense.

Now we finally come to the soundness statement for verify:

$$\mathsf{verify\text{-}sound} : \forall\ (S : \mathsf{Sequent})\ (p : \mathsf{Proof}) \to$$
$$\mathsf{good}\ S \to \mathsf{T}\ (\mathsf{verify}\ S\ p) \to \mathsf{unsat}\ S$$

The condition good S is necessary, because the soundness of TESC proofs is dependent on the invariant that all sequents are good. 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. parse-verify is the outer function which accepts two character lists as argument, parses them into a Sequent and a Proof, and calls verify. The soundness statement for parse-verify is as follows:

$$\begin{array}{l}\mathsf{parse\text{-}verify\text{-}sound} : \forall\ (\mathit{seq\text{-}chars}\ \mathit{prf\text{-}chars} : \mathsf{Chars}) \rightarrow \\ \quad \mathsf{T}\ (\mathsf{parse\text{-}verify}\ \mathit{seq\text{-}chars}\ \mathit{prf\text{-}chars}) \rightarrow \\ \quad \exists\ \lambda\ (S : \mathsf{Sequent}) \rightarrow \mathsf{returns}\ \mathsf{parse\text{-}sequent}\ \mathit{seq\text{-}chars}\ S \times \mathsf{unsat}\ S\end{array}$$

parse-verify-sound is an improvement over verify-sound, but it also shows the limitation of the current setup. We know that there is *some* unsatisfiable Sequent parsed from the input characters, but we have no guarantees that this sequent is actually equivalent to the original TPTP file. This means that the formal verification of VTV is limited to the soundness of its proof-checking kernel, and the correctness its TPTP parsing stage has to be taken in faith.

7 Test Results

The performance of VTV was tested by running it on all elligible problems in the TPTP [] problem library. A TPTP problem is elligible 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 (23291).
- Its status is 'theorem' or 'unsatisfiable' (13636).
- It conforms to the official TPTP syntax. More precisely, it does not have any occurrences of the charater '%' 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).

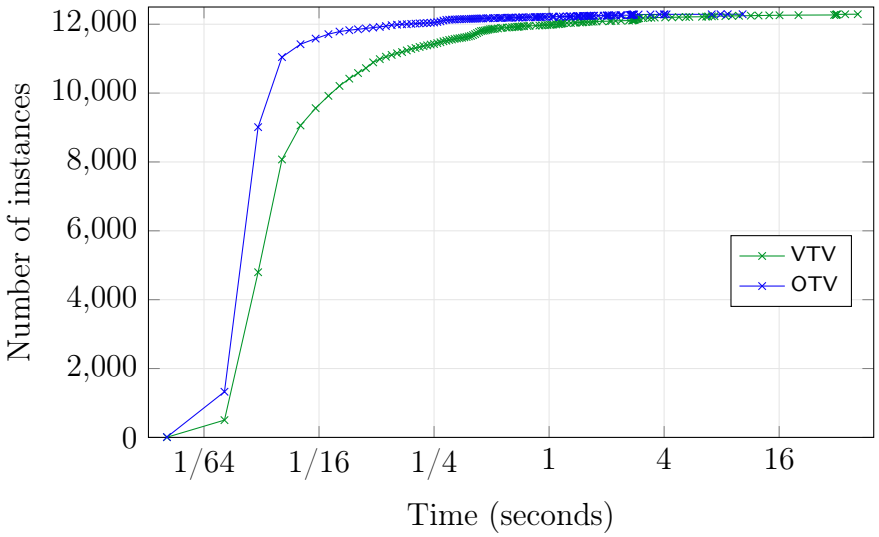

Figure 1: Verification times of VTV and DTV. The datapoints show the number of TESC proofs that each verifier could check within the given time limit. The plots look more "jumpy" toward the lower end due to the limited time measurement resolution (10 ms) of the unix `time` command.

- It can be solved by Vampire or E in one minute using default settings (Vampire = 7885, E = 4853).
- 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. For more information on the exact testing setup, refer to...

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 the T3P tool running on the same proofs. The default TESC verifier is written in Rust, and is optimized for performance with no regard to verification. For convenince, we refer to it as the Optimized TESC Verifier (OTV).

VTV is slower than OTV as expected, but the difference is unlikely to be

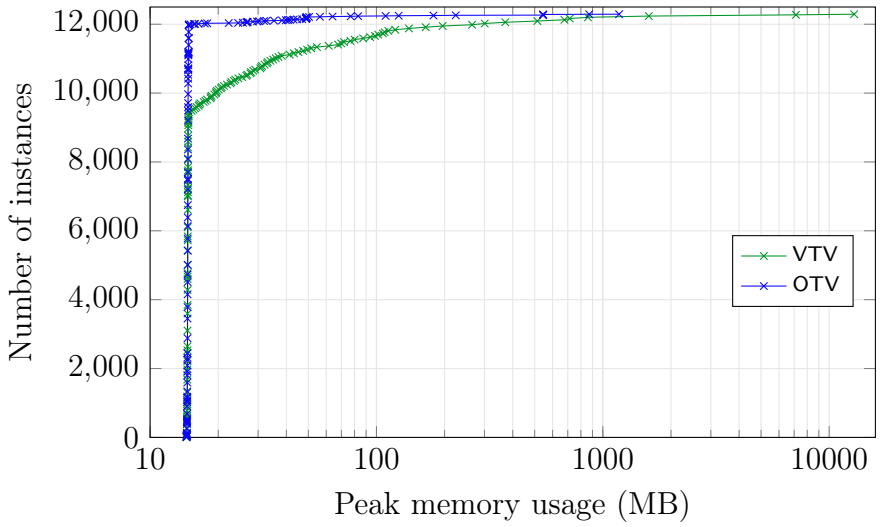

Figure 2: Peak memory usages of VTV and DTV. The datapoints of show the number of TESC proofs that each verifier could check within the given peak memory usage.

noticed in actual use since the total times are dominated by a few outliers and the absolute times for most proofs are very short: the median time for VTV is 40 ms, which is only 10 ms behind the OTV's 30 ms. Also, VTV verified >97.4% of proofs under 1 second, and >99.3% under 5 seconds. But OTV's mean time (54.54 ms) is still much shorter than that of VTV (218.93 ms), so users may prefer OTV for verifying one of the hard outliers or processing a batch large 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 usage for both verifiers are stable and stays within the 14-20 MB range, but VTV spikes earlier and higher than OTV. Due 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 we may assume that the memory requirement for verifying all 12296 proofs with VTV is >64 GB. In contrast, OTV could verify all 12296 proofs with less than 3.2 GB of peak memory usage.

8 Conclusion

VTV increases our confidence in the correctness of TESC proofs by providing an alternative that users can fall back on whenever extra rigour is required. It also helps the design of other TESC verifiers by serving as a reference implementation that is guaranteed to be correct. It should be particularly helpful for implementing other verified TESC verifiers in, say, Lean or Coq, since many of the issues we've discussed (termination checking, data structures, etc.) are shared between these languages.

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 finer low-level control over memory usage, e.g. ATS.

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, making VTV return the parsed problem in a temporary file and textually comparing it with the original problem file.

References

[1] 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.