A Formally Verified TESC Verifier

Seulkee Baek

30th January, 2021

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 elligible problems from the TPTP problem library, successfully verifying all but the largest 5 of 12296 proofs, with >97% of the proofs verified under 1 second.

1 Introduction

Modern automated reasoning tools are highly complex software whose correctness is no simple matter to establish. Bugs have been discovered in them in the past [16, 11], 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 [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 themselves. 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 [15] 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 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 primary purpose is to demonstrate the reliability of TESC proofs by showing precisely what is established by their successful verification using VTV. Its secondary aim is to serve as a guide for understanding VTV's codebase and its design choices, which can be especially useful for readers who want to implement their own TESC verifiers.

The rest of the paper is organized as follows: Section 2 gives a brief survey of simlar works and how VTV relates to them. Section 4 describes the syntax and inference rules of the TESC proof calculus. Section 5 presents the main TESC verifier kernel, and Section 6 gives a detailed specification of the verifier's soundness property. Sections 4, 5, and 6 also include code excerpts and discuss how their respective contents are formalized in Agda. Section 7 shows the results of empirical tests measuring VTV's performance. Section 8 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 [7], and GRIT [8]. 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 self-verification of interactive theorem prover kernels. All of HOL Light except the axiom of infinity has been proven consistent in HOL Light itself [11], which allows us to consider HOL Light consistent for most proofs in practice. More recent work on the Metamath Zero [4] theorem prover aims to not only verify a large part of Metamath Zero's logic in itself, but also its implementation down to the level of 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 to verifying the output of ATPs, but (to the extent of my knowledge) none has used a proof format with a verified checker. For instance, GDV [17] works by breaking down an ATP-generated solution into small subproblems and re-solving them with multiple unverified ATPs. Foundational Proof Certificates [6] is a system that could be used to specify proof formats and implement proof checkers for first-order ATPs, but its actual implementation [5] has been limited to a small subset of inferences used by ATPs 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

The syntax of the TESC calculus is as follows:

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

f ranges over *functors*, which are usually called 'non-logical symbols.' 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.

t ranges 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 is its De Bruijn index.

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 the *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. Note that:

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

Table 1: TESC inference rules.

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

Table 2: Formula analysis functions. For any arguments not explicitly defined above, the functions all return $\top$. E.g., $A(0, \phi \vee \psi) = \top$.

- $\Gamma[i]$ denotes the (0-based) ith formula of sequent Γ, where $\Gamma[i] = \top$ if the index i is out-of-bounds.

- $\text{lfi}(x)$ denotes the largest functor index (lfi) occuring in x. If x incudes no functor indices, $\text{lfi}(x) = -1$.

- $\text{size}(\Gamma)$ is the number of formulas in Γ.

- All inference rules are designed to preserve the invariant $\text{lfi}(\Gamma) < \text{size}(\Gamma)$ for every sequent Γ. We say that a sequent Γ is *good* if it satisfies this invariant.

- S is the usual cut rule, and X is the axiom or init rule.

- $\text{adm}(k, \phi)$ asserts that ϕ is an *admissable* formula in respect to the target theory and sequent size k. More precisely, if T is the target theory, Γ is a sequent satisfiable modulo T, and $\text{size}(\Gamma) = k$, then $\text{adm}(k, \phi)$ implies that adding ϕ to Γ preserves satisfiability modulo T. The sequent size argument k is required becuase some admissable formulas use this number as the functor index of newly introduced indexed functors. The T rule may be used to add any admissable formula.

The last part implies that the definition of the admissability predicate, and by extension the definition of well-formed TESC proofs, changes according to 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 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
```

$$\begin{aligned}
&\textsf{rule-t} : \textsf{Formula} \to \textsf{Proof} \to \textsf{Proof} \\
&\textsf{rule-x} : \textsf{Nat} \to \textsf{Nat} \to \textsf{Proof}
\end{aligned}$$

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 The 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. This is exactly what verify does:

$$\begin{aligned}
&\textsf{verify} : \textsf{Sequent} \to \textsf{Proof} \to \textsf{Bool} \\
&\textsf{verify}\ \Gamma\ (\textsf{rule-a}\ i\ b\ p) = \textsf{verify}\ (\textsf{add}\ \Gamma\ (\textsf{analyze-a}\ b\ (\textsf{nth}\ i\ \Gamma)))\ p \\
&\textsf{verify}\ \Gamma\ (\textsf{rule-b}\ i\ p\ q) = \\
&\quad (\textsf{verify}\ (\textsf{add}\ \Gamma\ (\textsf{analyze-b}\ \textsf{false}\ (\textsf{nth}\ i\ \Gamma)))\ p) \wedge \\
&\quad (\textsf{verify}\ (\textsf{add}\ \Gamma\ (\textsf{analyze-b}\ \textsf{true}\ (\textsf{nth}\ i\ \Gamma)))\ q) \\
&\textsf{verify}\ \Gamma\ (\textsf{rule-c}\ i\ t\ p) = \\
&\quad \textsf{term*-lfi<?}\ (\textsf{suc}\ (\textsf{size}\ \Gamma))\ t \wedge \\
&\quad \textsf{verify}\ (\textsf{add}\ \Gamma\ (\textsf{analyze-c}\ t\ (\textsf{nth}\ i\ \Gamma)))\ p \\
&\textsf{verify}\ \Gamma\ (\textsf{rule-d}\ i\ p) = \textsf{verify}\ (\textsf{add}\ \Gamma\ (\textsf{analyze-d}\ (\textsf{size}\ \Gamma)\ (\textsf{nth}\ i\ \Gamma)))\ p \\
&\textsf{verify}\ \Gamma\ (\textsf{rule-n}\ i\ p) = \textsf{verify}\ (\textsf{add}\ \Gamma\ (\textsf{analyze-n}\ (\textsf{nth}\ i\ \Gamma)))\ p \\
&\textsf{verify}\ \Gamma\ (\textsf{rule-s}\ \phi\ p\ q) = \\
&\quad \textsf{formula-lfi<?}\ (\textsf{suc}\ (\textsf{size}\ \Gamma))\ \phi \wedge \\
&\quad \textsf{verify}\ (\textsf{add}\ \Gamma\ (\textsf{not}\ \phi))\ p \wedge \textsf{verify}\ (\textsf{add}\ \Gamma\ \phi)\ q \\
&\textsf{verify}\ \Gamma\ (\textsf{rule-t}\ \phi\ p) = \\
&\quad \textsf{adm?}\ (\textsf{size}\ \Gamma)\ \phi \wedge \textsf{verify}\ (\textsf{add}\ \Gamma\ \phi)\ p \\
&\textsf{verify}\ \Gamma\ (\textsf{rule-x}\ i\ j) = \textsf{formula=?}\ (\textsf{nth}\ i\ \Gamma)\ (\textsf{not}\ (\textsf{nth}\ j\ \Gamma))
\end{aligned}$$

Analytic rules introduce new formulas obtained by formula analysis using analyze functions, and side conditions are checked using appropriate ? 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, 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) tool 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 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 b 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 Soundness

In order to verify 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 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

$$\begin{aligned}&\mathsf{Rels} : \mathsf{Set}\\&\mathsf{Rels} = \mathsf{List}\ D \to \mathsf{Bool}\end{aligned}$$

$$\begin{aligned}&\mathsf{Funs} : \mathsf{Set}\\&\mathsf{Funs} = \mathsf{List}\ D \to D\end{aligned}$$

A Rels (resp. Funs) can be thought of as a collection of an infinite number of relations (resp. functions), one for each arity. A interpretation is an assignments of Rels and Funs to functors.

$$\begin{aligned}&\mathsf{RA} : \mathsf{Set}\\&\mathsf{RA} = \mathsf{Functor} \to \mathsf{Rels}\end{aligned}$$

$$\begin{aligned}&\mathsf{FA} : \mathsf{Set}\\&\mathsf{FA} = \mathsf{Functor} \to \mathsf{Funs}\end{aligned}$$

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

$$\begin{aligned}&\mathsf{VA} : \mathsf{Set}\\&\mathsf{VA} = \mathsf{Nat} \to D\end{aligned}$$

We can now define the valuation of terms and forms under interpretations and variable assignments. Term valuation requires a bit of ingenuity due to the unusual definition of `Term*`:

$$\begin{aligned}&\mathsf{ElemList} : \mathsf{Set} \to \mathsf{Bool} \to \mathsf{Set}\\&\mathsf{ElemList}\ A\ \mathsf{false} = A\end{aligned}$$

$$\mathsf{ElemList}\ A\ \mathsf{true} = \mathsf{List}\ A$$

$$
\begin{aligned}
&\mathsf{term^*\text{-}val} : \mathsf{FA} \to \mathsf{VA} \to \{b : \mathsf{Bool}\} \to \mathsf{Term^*}\ b \to \mathsf{ElemList}\ D\ b\\
&\mathsf{term^*\text{-}val}\ \_\ V\ (\mathsf{var}\ k) = V\ k\\
&\mathsf{term^*\text{-}val}\ F\ V\ (\mathsf{fun}\ f\ ts) = F\ f\ (\mathsf{term^*\text{-}val}\ F\ V\ ts)\\
&\mathsf{term^*\text{-}val}\ F\ V\ \mathsf{nil} = []\\
&\mathsf{term^*\text{-}val}\ F\ V\ (\mathsf{cons}\ t\ ts) = (\mathsf{term^*\text{-}val}\ F\ V\ t) :: (\mathsf{term^*\text{-}val}\ F\ V\ ts)
\end{aligned}
$$

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

$$
\begin{aligned}
&\_,\_,\_\vDash\_ : \mathsf{RA} \to \mathsf{FA} \to \mathsf{VA} \to \mathsf{Formula} \to \mathsf{Set}\\
&R\ ,\ F\ ,\ V \vDash (\mathsf{cst}\ b) = \mathsf{T}\ b\\
&R\ ,\ F\ ,\ V \vDash (\mathsf{not}\ \phi) = \neg\ (R\ ,\ F\ ,\ V \vDash \phi)\\
&R\ ,\ F\ ,\ V \vDash (\mathsf{bct}\ \mathsf{or}\ \phi\ \psi) = (R\ ,\ F\ ,\ V \vDash \phi) \uplus (R\ ,\ F\ ,\ V \vDash \psi)\\
&R\ ,\ F\ ,\ V \vDash (\mathsf{bct}\ \mathsf{and}\ \phi\ \psi) = (R\ ,\ F\ ,\ V \vDash \phi) \times (R\ ,\ F\ ,\ V \vDash \psi)\\
&R\ ,\ F\ ,\ V \vDash (\mathsf{bct}\ \mathsf{imp}\ \phi\ \psi) = (R\ ,\ F\ ,\ V \vDash \phi) \to (R\ ,\ F\ ,\ V \vDash \psi)\\
&R\ ,\ F\ ,\ V \vDash (\mathsf{bct}\ \mathsf{iff}\ \phi\ \psi) = (R\ ,\ F\ ,\ V \vDash \phi) \leftrightarrow (R\ ,\ F\ ,\ V \vDash \psi)\\
&R\ ,\ F\ ,\ V \vDash (\mathsf{qtf}\ \mathsf{false}\ \phi) = \forall\ x \to (R\ ,\ F\ ,\ (V\ /\ 0 \mapsto x) \vDash \phi)\\
&R\ ,\ F\ ,\ V \vDash (\mathsf{qtf}\ \mathsf{true}\ \phi) = \exists\ \lambda\ x \to (R\ ,\ F\ ,\ (V\ /\ 0 \mapsto x) \vDash \phi)\\
&R\ ,\ F\ ,\ V \vDash (\mathsf{rel}\ r\ ts) = \mathsf{T}\ (R\ r\ (\mathsf{term^*\text{-}val}\ F\ V\ ts))
\end{aligned}
$$

`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:

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

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

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

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:

$$\begin{array}{l} \textsf{verify-sound} : \forall\ (S : \textsf{Sequent})\ (p : \textsf{Proof}) \rightarrow \\ \quad \textsf{good}\ S \rightarrow \textsf{T}\ (\textsf{verify}\ S\ p) \rightarrow \textsf{unsat}\ S \end{array}$$

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:

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

parse-verify-sound is an improvement over verify-sound, but it also shows the limitation of the current setup. It asserts 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).
- 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 noticed in actual use since the total times are dominated by a few outliers

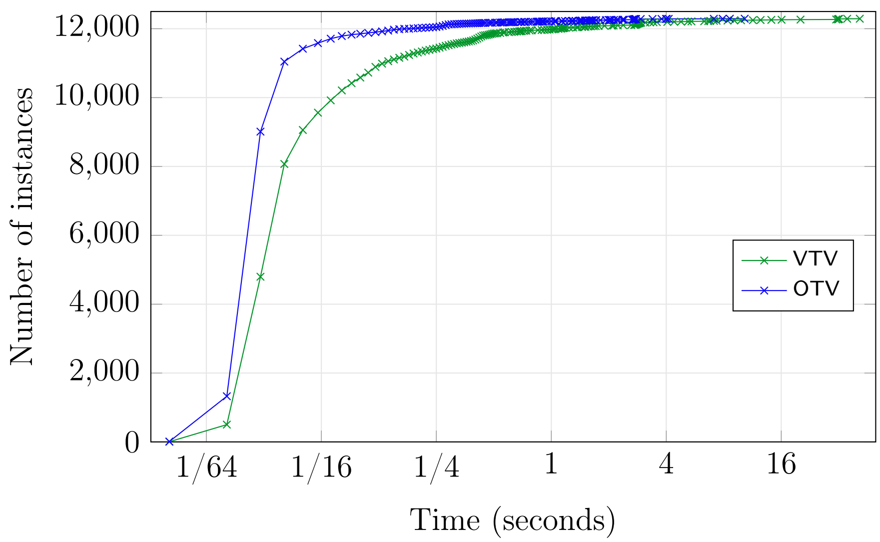

Figure 1: Verification times of VTV and DTV. The datapoints show the number of TESC proofs that each verifier could check within the given time limit. The plots look more "jumpy" toward the lower end due to the limited time measurement resolution (10 ms) of the unix `time` command.

and the absolute times for most proofs are very short: the median time for VTV is 40 ms, a mere 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 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 usage for both verifiers are stable and stays within the 14-20 MB range, but VTV's memory usage spikes both 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 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.

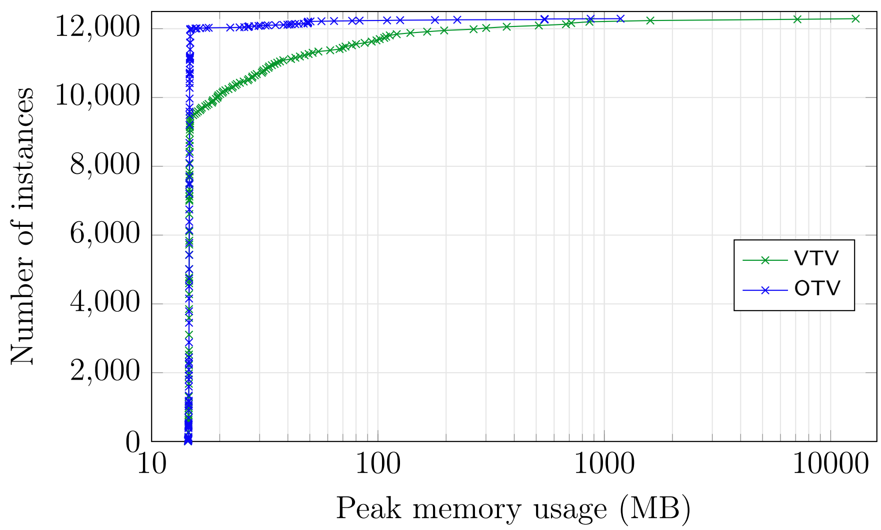

Figure 2: Peak memory usages of VTV and DTV. The datapoints of show the number of TESC proofs that each verifier could check within the given peak memory usage.

8 Conclusion

VTV can serve as a fallback option whenever extra rigour is required in verification, thereby increasing our confidence in the correctness of TESC proofs. It also helps the design of other TESC verifiers by providing 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 common to 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.

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 with the original problem.

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] Zakaria Chihani, Dale Miller, and Fabien Renaud. Foundational proof certificates in first-order logic. In *International Conference on Automated Deduction*, pages 162–177. Springer, 2013.

[7] 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.

[8] 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.

[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] Giles Reger and Martin Suda. Checkable proofs for first-order theorem proving. In *ARCADE@ CADE*, pages 55–63, 2017.

[16] 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.

[17] Geoff Sutcliffe. Semantic derivation verification: Techniques and implementation. *International Journal on Artificial Intelligence Tools*, 15(06):1053–1070, 2006.