The TESC Proof Format for First-Order ATPs

Seulkee Baek

Department of Philosophy, Carnegie Mellon University

seulkeeb@andrew.cmu.edu

Abstract

TESC is a machine-checkable, low-level proof format for first-order ATPs. Every step in a TESC proof is verifiable, including preprocessing steps like Skolemization and CNF normalization. TESC proofs are processed using the T3P tool, which generates TESC proofs from input TSTP solutions and checks them with multiple backend verifiers for redundancy. TESC and T3P demonstrated robust performance in a test using all eligible problems in the TPTP library, successfully compiling 98.2% (resp. 92.8%) of TSTP solutions produced by Vampire (resp. E) and verifying all proofs in 3.27% (resp. 2.51%) of the time it takes to produce their corresponding TSTP solutions.

1 Introduction

The lack of a standard, machine-checkable proof format is a long-standing issue for first-order automated theorem provers (ATPs). Keeping a modern ATP bug-free while introducing new features is notoriously difficult [], and the increasing complexity of ATPs only makes it more risky to trust them without checkable proofs. In addition, the lack of proofs also makes it difficult to *use* the output of ATPs; e.g., there are many proof assistants which solve first-order goals by calling external ATPs, but they are forced to either blindly trust the ATPs' answers [] or perform redundant proof searches [] because the ATP output cannot be directly translated into valid proofs for the proof assistants.

These limitations make it clearly desirable to introduce an ATP proof format which lifts the burden of paintaking correctness checks from ATP development

and improves the interface between ATPs and their client software. The five main desiderata for for a prospective ATP proof format, as listed by Martin and Suda [], are as follows:

1. Compatibility with all inference techniques used by major ATPs
2. Coverage of "unsound" steps whose conclusions are not logically implied by their premises
3. Efficiency of verification
4. Ease of implementation in ATPs with small performance overhead
5. Wide adoption by the ATP commmunity

The Theory Extensible Sequent Calculus (TESC) is a new proof format for first-order ATPs that aims to fill this gap, with a particular emphasis on 2, 3, and 4. It supports 2 by including rules for introducing an extensible set of theory axioms, and ensures 3 by compiling high-level proofs down to a fully elaborated low-level format. 4 is achieved by using TSTP files which most ATPs are already capable of generating, so it requires no implementation effort and acceptably small performance penalties. It is hoped that these strengths will encourage adoption by ATP users and developers, leading to 5. 1 is also important in the long run, but it is not the focus of this work as it has lower immediate priority.

The following sections are organized as follows: section 2 discusses how this project relates to existing work. Section **??** describes the conventions used throughout the paper. Section 3 presents the underlying proof calculus of the format. Section 4 gives the concerete TESC syntax. Section 6 discusses the T3P tool for generating and verifying TESC files. Section 5 shows a simple example of a TPTP problem and its TESC proof. Section 7 presents the results of tests using problems from the TPTP library. Section 8 summarizes what was accomplished and discusses future work.

2 Related Works

The aforementioned survey by Martin and Suda [] provides a concise summary of the status quo regarding ATP proof formats. The testing and verification

efforts of the Vampire development team [] gives a good sense of the difficulty and importance of ensuring correctness for cutting-edge ATP systems.

The standardization of ATP input and output offered by the TPTP [] and TSTP [] formats is a critical prerequisite which makes the TESC format possible.

The largest difference between TESC and existing approaches to ATP result verification is that TESC has direct support for preprocessing steps (e.g. Skolemization) whose conclusions are not logically implied by existing premises. TESC is most closely related to Ivy [] and GDV [], and is especially similar to the former in that it uses ATPs to generate independently verifiable proof objects. But both Ivy and GDV have limited support for preprocessing steps: GDV omits their verification entirely, and Ivy prevents ATPs from using optimized preprocessing strategies because it relies on a formally verified preprocessor.

Outside first-order logic, SAT solvers also used to suffer from a similar lack of standard checkable proof formats. This problem was solved by the introduction of DRAT [5] and LRAT [2] formats, whose convincing success was a direct inspiration for the development of the TESC format.

3 Proof Calculus

The syntax of terms and formulas for the TESC proof calculus is shown in Figure 1. We let f range over *functors*, which are used as both function and relation symbols. TESC makes no distinction bewteen the two types of symbols because it simplifies implementation and is otherwise harmless, as the type of a symbol can always be determined from its context. The name 'functor' is borrowed from the TPTP syntax terminology.

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

Figure 1: Abstract syntax of TESC terms and formulas.

The symbol σ ranges over *plain functors*, which are the usual relation or function symbols of first-order logic. We assume that there is a suitable set of symbols Σ, and let $\sigma \in \Sigma$. Symbols of the form $\#_k$ are *indexed functors*, where the number k is called the *functor index* of $\#_k$. Indexed functors are useful for reducing the cost of introducing fresh functors: if you know 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. [1]

We let t to range over terms, $\vec{t}$ over lists of terms, ϕ over formulas, and Γ over sequents. The number k in variable x_k is the variable's *De Bruijn index* [3]; the use of De Bruijn indices allows quantifiers to be written without accompnying variables. E.g., the variable x_1 in formula $\forall\exists P(x_0, x_1)$ is bound by $\forall$. 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. For common binary predicates we take the liberty of using infix notations, e.g. writing $x = y$ instead of $=(x, y)$.

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 [4], which is where they get their names from.

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

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

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

Table 1: TESC inference rules. $\mathrm{size}(\Gamma)$ 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. $\mathrm{adm}(k, \phi)$ asserts that ϕ is an admissable formula in respect to the target theory and sequent size k.

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 size(Γ) = k, the sequent Γ, ϕ is also satisfiable modulo **T**.

More intuitively, the T rule allows you 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. In other words, TESC is closer to a *scheme* for proof formats than to a single, fixed proof format: users can obtain specific *implementations* of TESC by defining the adm predicate, thereby fixing the set of admissable formulas that may be introduced via the T rule. The implementation must also include a decision procedure that determines the truth of adm(k, ϕ) for any input value of k and ϕ in order to make the resulting format mechanically verifiable.

When there is a need to distinguish the general scheme from the concrete instance, we refer to the current implementation of TESC as TESC-0. In TESC-0, adm(k, ϕ) holds if and only if ϕ is either a trivial logical constant ($\top$ or $\neg bot$), one of the three equality axioms (reflexivity, symmetry, or transitivity), or has one of the following forms:

- $\forall\forall(x_1 = x_0 \rightarrow \ldots \forall\forall(x_1 = x_0 \rightarrow f(x_{2k+1}, \ldots, x_1) = f(x_{2k}, \ldots, x_0))\ldots)$, where f is any functor and $2k$ is the total number of $\forall$s
- $\forall\forall(x_1 = x_0 \rightarrow \ldots \forall\forall(x_1 = x_0 \rightarrow (r(x_{2k+1}, \ldots, x_1) \rightarrow r(x_{2k}, \ldots, x_0)))\ldots)$, where r is any functor and $2k$ is the total number of $\forall$s
- $\forall \ldots \forall(\#_k(x_0, \ldots, x_m) \leftrightarrow \phi)$, where ϕ is any formula and m is the number of outer universal quantifiers $\forall \ldots \forall$.
- $\forall \ldots \forall(\exists\phi \rightarrow \phi[0 \mapsto \#_k(x_0, \ldots, x_m)])$, where ϕ is any formula and m is the number of outer universal quantifiers $\forall \ldots \forall$.

The first two takes care of congruence, the third handles new predicate definitions, and the last is used for adding choice axiom. Note that TESC-0 has no special support for Skolemization other than the choice axioms. Therefore, any Skolemization step that is not an instance of the choice axiom given above (e.g., simultaneous elimination of multiple existential quantifiers) must be 'spelled out' as the introduction and usage of multiple choice axioms.

Let us say that a sequent Γ is *good* if it satisfies $\mathrm{lfi}(\Gamma) < \mathrm{size}(\Gamma)$, where $\mathrm{lfi}(x)$ denotes the largest functor index occurring in an expression x (if x incudes no functor indices, $\mathrm{lfi}(x) = -1$). All TESC inference rules are designed to preserve goodness; i.e., if the conclusion sequent of a rule is good, then so are all of its premise sequents. (we are concerned with preservation in the bottom-up direction, since this is the direction in which proofs are constructed and verified). Most rules (A, B, N, and X) trivially preserve goodness since they do not introduce any new terms or formulas. Others preserve goodness either by carefull choosing the newly introduced term (D) or enforcing it as a side condition (C, S, and T). The invariant that all sequents are good allows you to use the indexed functor $\#_{\mathrm{len}(\Gamma)}$ whenever a fresh functor is needed for an inference (e.g., new predicate definitions).

4 The Format

The complete concrete syntax of the TESC proof format is given in Figure 2. The syntax presentation is similar to that of the TPTP syntax.

The mapping between TESC syntax and the abstract syntax used in Section 3 is self-explanatory except for `<proof>`. In a `<proof>`, the first character always indicates the inference rule to be applied, followed immediately by arguments that determine how that inference rule should be applied, which in turn are followed by any remaining subproofs. E.g., in order to apply the A rule, we need a boolean argument b and a number i in order to determine the new formula $A(b, \Gamma[i])$ to be added, so the leading character 'A' is followed by `<number>` and `<bool>`. For more details on how arguments for inference rule applications are read and used, see Section 4.

One useful feature of the TESC syntax is that it is completely backtracking-free, because at any given point during parsing, each new character read unambiguously determines the syntactic class being parsed. This not only simplifies parser design, but also allows streaming verification where the verifier immediately parses and checks the prefix of proof already read without waiting for the remaining suffix.

```
<hash> ::: [#]                <percent> ::: [%]
<dot> ::: [.]                 <dollar> ::: [$]
<comma> ::: [,]               <not_percent> ::: [^%]
<tilde> ::: [~]               <zero_numeric> ::: [0]
<vline> ::: [|]               <non_zero_numeric> ::: [1-9]
<ampersand> ::: [&]           <numeric> ::: [0-9]
<arrow> ::: [>]               <boolean> ::: [01]
<equal> ::: [=]               <exclamation_mark> ::: [!]
<double_quote> ::: ["]        <question_mark> ::: [?]

<positive_decimal> ::= <non_zero_numeric><numeric>*
<decimal> ::= <zero_numeric> | <positive_decimal>
<number> ::= <decimal><percent>
<string> ::= <not_percent>*<percent>
<plain_functor> ::= <double_quote><string>
<indexed_functor> ::= <hash><number>
<functor> ::= <plain_functor> | <indexed_functor>
<variable> ::= <hash><number>
<term> ::= <variable> | <dollar><functor><term_list>
<term_list> ::= <dot> | <comma><term><term_list>
<connective> ::= <vline> | <ampersand> | <arrow> | <equal>
<quantifier> ::= <exclamation_mark> | <question_mark>
<formula> ::= <boolean>
            | <dollar><functor><term_list>
            | <tilde><formula>
            | <connective><formula><formula>
            | <quantifier><formula>
<proof> ::= A<number><boolean><proof>
          | B<number><proof><proof>
          | C<number><term><proof>
          | D<number><proof>
          | N<number><proof>
          | S<formula><proof><proof>
          | T<formula><proof>
          | X<number><number>
```

Figure 2: TESC syntax

5 A Simple Example

For a simple example of a TESC proof, consider the following problem (problem COL086-1 from the TPTP library):

```
cnf(fond_bird_exists,hypothesis,
    ( response(a,b) = b )).

cnf(prove_happiness_2,negated_conjecture,
    (  response(a,B) != A )).
```

A valid TESC proof for the problem, with line breaks and indentation added for legibility, is as follows:

```
C1%$"a%.
  C2%$"response%,$"a%.,$"a%..
    T!$"=%,#0%,#0%.
      C4%$"response%,$"a%.,$"a%..
        X3%5%
```

Here's the same proof presented in a more human-readable form, where irrelevant formulas are abbreviated and Γ is the initial sequent that contains formulas in the input problem:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma,\ldots,=(\text{response(a, a)},\text{response(a, a)})}\,X(3,5)}{\Gamma,\ldots,\forall=(x_0,x_0)}\,C(4,\text{response}(a,a))}{\Gamma,\ldots,\neg=(\text{response(a, a)},\text{response(a, a)})}\,T(\forall=(x_0,x_0))}{\Gamma,\forall\neg=(\text{response(a, a)},x_0)}\,C(2,\text{response}(a,a))}{\Gamma}\,C(1,a)$$

6 The T3P Tool

The TPTP-TSTP-TESC Processor (T3P) is the main tool for generating and verifying TESC proofs. T3P accepts a TPTP problem and its TSTP solution as input, and compiles them to a TESC proof. Due to the wide variation of TSTP outputs between ATPs, there cannot be a universal TSTP-to-TESC compiler, and proof compilation must be implemented separately for each ATP. T3P currently supports compilation of solutions produced by Vampire and E in monomorphic first-order logic with equality.

Since TESC requires explicit detail for all of its inferences, T3P must perform searches to fill in the information missing from TSTP solutions, such as positions of rewritten terms or pivot literals in resolution steps. These searches may occasionally blow up and cause compilation failures. For some types of inferences, search space explosion can be mitigated with custom-tailored solutions. E.g., T3P handles Vampire's AVATAR steps by exporting the problem to CaDiCaL [1] and translating its proof output back to TESC. But many more complex inferences cannot be handled this way, so the best way to ensure fast and reliable compilation to TESC is generating fine-grained TSTP solutions with detailed information.

T3P also accepts a TPTP problem and its TESC proof as input, and verifies that the latter is a valid proof of the former's unsatisfiability. T3P includes three different backend verifiers written in Rust, Agda, and Prolog that users can choose from. T3P uses the Rust verifier by default, since it is optimized for performance and is most suitable for practical proof checking. The Agda verifier comes with a formal specification and proof of its soundness (whose details will be discussed in a separate upcoming paper) and is reasonably performant for most proofs, so it can serve as a fallback option when extra reliability is needed. The Prolog verifier is used mainly for debugging purposes.

Algorithm 1 shows a high-level description of a TESC verifier. Note that:

- The argument *prob* is a list that holds the formulas of the input TPTP problem, *in the order* they originally appear in that problem. If a problem has `include` clauses, the contents of included files are treated as inlined in the position of their `include` clauses. The ordering is important because TESC inference rules identify formulas by their positions in sequents.

- The use of *prob* implies that TPTP formulas must be parsed into TESC formulas before calling the main Verify function. This parsing is mostly straightforward, but there are a couple caveats:

  - ⋆ TPTP formulas in the CNF language are implicitly universally quantified, so the parser must choose some ordering for the variables in a CNF formula before performing universal closure. This ordering may be arbitrary but must be consistent, since a TESC proof generated assuming one ordering and verified using another will fail to check. T3P sorts variables in a CNF formula by their order of appearance in a left-to-write sweep: the earlier its appearance in the

sweep, the lower its De Bruijn index.

  - ⋆ The TPTP syntax includes logical connectives whose direct equivalents do not exist in TESC (e.g., <~>). During parsing, they are replaced by suitable (combinations of) TESC connectives that preserve logical equivalence.

- The argument *prf* is the list of characters in the input TESC proof.
- The Verify function returns 1 if the proof is valid, and 0 otherwise.
- For each datatype X, the fuction call ReadX(*prf*) consumes as many characters as necessary from *prf* to parse and return a term of type X. This also implicitly modifies *prf* as the consumed characters are removed.
- Some functions used in Verify may fail. E.g., ReadBool(*prf*) will fail if the first character of *prf* is neither '0' nor '1'. We assume that the caller of Verify implements explicit failure handling such that, if Verify fails at any point, it is handled in the same way as when Verify returns 0.
- Pop and Push adds an element to or removes an element from the beginning of a list.

Algorithm 1 TESC verification

```
1:  function VERIFY(prob, prf)
2:      sqts ← [prob]
3:      while sqts ≠ [ ] do
4:          sqt ← Pop(sqts)
5:          c ← Pop(prf)
6:          case c of
7:              'A' ⇒
8:                  i ← ReadNumber(prf)
9:                  b ← ReadBool(prf)
10:                 f ← ApplyA(b, sqt[i])
11:                 Push(sqt ++ [f], sqts)
12:             ...          ▷ Other analytic rules are similar to A
13:             'S' ⇒
14:                 f ← ReadFormula(prf)
15:                 Assert(LargestFunctorIndex(f) ≤ Length(sqt))
16:                 Push(sqt ++ [f], sqts)
17:                 Push(sqt ++ [¬f], sqts)
18:             'T' ⇒
19:                 f ← ReadFormula(prf)
20:                 Assert(LargestFunctorIndex(f) ≤ Length(sqt))
21:                 Assert(Admissible(f))
22:                 Push(sqt ++ [f], sqts)
23:             'X' ⇒
24:                 i ← ReadNumber(prf)
25:                 j ← ReadNumber(prf)
26:                 Assert(sqt[i] = ¬sqt[j])
27:             _ ⇒ return 0
28:     _ ⇒ return 1
```

7 Test Results

The performance of TESC and T3P was tested using all eligible problems from the TPTP problem library. A given TPTP problem is eligible if it is (1) in the CNF or FOF languauge, (2) has 'theorem' or 'unsatisfiable' status, (3) conforms to the official TPTP syntax, (4) assigns unique names to all of its formulas, and (5) is solved by Vampire or E under one minute. (3) is necessary because TESC and T3P assumes (per the official TPTP syntax) that the character '%' does not appear in the `sq_char` syntactic class, and uses it as an endmarker when parsing `sq_char`. (4) needs to be enforced in order to let T3P to unambiguously refer to premises during compilation. There were a total of 7885 eligible problems for Vampire, and 4853 for E.

Add system information

Using the eligible problems, T3P successfully compiled 7792 and 4504 TSTP solutions generated by Vampire and E, respectively. All proofs produced were verified by T3P. The difference between compilation success rates for Vampire (98.8%) and E (92.8%) can be largely attributed to solution granularity: TSTP solutions produced by E often chain together multiple inferences into a single step and omit the intermediate formulas, so T3P has to perform more search to fill in the missing information.

Table 2 shows the test results for the problems whose solutions were successfully compiled by T3P. The execution time results are encouraging, as it only takes a 39-67% overhead to produce a checkable proof compared to the base ATP solution time. Verification times are practically negligible compared to solution times, which shows that TESC is efficient for re-use of ATP search results. The main downside of TESC is the >8 times increase in file size compared to TSTP, but this is unlikely to be a serious problem since ATP proofs are typically small (note that the set of *all* TESC proofs that could be generated for the TPTP library still fits under 7 GBs) and ATPs usually exhaust time/memory limits well before producing excessively large proofs.

	Vampire	E
Solution time (s)	30220.478	8473.055
TSTP solutions size (MB)	735.981	53.841
Compilation time (s)	11796.303	5701.754
Compilation time / solution time	0.390	0.673
TESC proofs size (MB)	6203.458	479.737
Proofs size / solutions size	8.429	8.910
Verification time (s)	989.180	212.990
Verification time / solution time	0.0327	0.0251

Table 2: Test results for TESC and T3P. Only the data for problems with successfully compiled TESC proofs are shown.

8 Conclusion

In this paper, we have introduced the TESC proof format and the T3P tool for generating and verifying TESC proofs. TESC imposes minimal overhead on ATPs by offloading difficult elaboration work to T3P. TESC also allows mechanical verification of SPO steps by reducing them to applications of a small number of well-defined T rules. In a comprehensive test using Vampire and E on all eligible TPTP problems, we have demonstrated that this approach scales to practical problems and state-of-the-art ATPs.

The main limitation of TESC at the moment is the compilation bottleneck: implementation of a TSTP-to-TESC compiler is highly time-consuming, must be duplicated for each ATP, and the resulting compiler is likely to require frequent updates because it is sensitive to small changes in the ATP output. The best solution to this problem is introducing a second intermediate format, most likely a subset of TSTP, that ATPs can generate directly and includes all the information required for efficient compilation. Using a single source format makes it easier to optimize and maintain a proof compiler, and its similarity to TSTP will also make it easier for developers to support its output in ATPs. Work on this intermediate format is already underway, and will be the next step in the development of TESC.

Acknowledgements

This work has been partially supported by AFOSR grant FA9550-18-1-0120.

References

[1] Armin Biere. Cadical at the sat race 2019. *SAT RACE 2019*, page 8.

[2] 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.

[3] 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.

[4] Raymond M Smullyan. *First-order logic*. Courier Corporation, 1995.

[5] Nathan Wetzler, Marijn JH Heule, and Warren A Hunt. Drat-trim: Efficient checking and trimming using expressive clausal proofs. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 422–429. Springer, 2014.

A Formula analysis functions

A	$A(0, \neg(\phi \vee \psi)) = \neg\phi$	$A(1, \neg(\phi \vee \psi)) = \neg\psi$
	$A(0, \phi \wedge \psi) = \phi$	$A(1, \phi \wedge \psi) = \psi$
	$A(0, \neg(\phi \rightarrow \psi)) = \phi$	$A(1, \neg(\phi \rightarrow \psi)) = \neg\psi$
	$A(0, \phi \leftrightarrow \psi) = \phi \rightarrow \psi$	$A(1, \phi \leftrightarrow \psi) = \psi \rightarrow \phi$
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	$D(k, \exists\phi) = \phi[0 \mapsto \#_k])$	$D(k, \neg\forall\phi) = \neg\phi[0 \mapsto \#_k]$
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. For the exact definition of substitutions using De Bruijn indices, see Appendix B.

B Subtitution for TESC terms and proofs

- $x_m[k \mapsto t] = \begin{cases} x_m, & \text{if } m < k \\ t, & \text{if } m = k \\ x_{m-1}, & \text{if } m > k \end{cases}$
- $(f(t_0, \ldots, t_m))[k \mapsto t] = f(t_0[k \mapsto t], \ldots, t_m[k \mapsto t])$
- If $C \in \{\top, \bot\}$, then $C[k \mapsto t] = C$
- $(\neg\phi)[k \mapsto t] = \neg(\phi[k \mapsto t])$
- If $\bullet \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$, then $(\phi \bullet \psi)[k \mapsto t] = (\phi[k \mapsto t]) \bullet (\psi[k \mapsto t])$,
- If $Q \in \{\forall, \exists\}$, then $(Q\phi)[k \mapsto t] = Q(\phi[k+1 \mapsto incr(t)])$, where $incr(t)$ is the result of incrementing all De Bruijn indices in t by 1
- $(r(t_0, \ldots, t_m))[k \mapsto t] = r(t_0[k \mapsto t], \ldots, t_m[k \mapsto t])$

Note that the incrementation/decrementation of De Bruijn indices are necessary to ensure that variables are bound to the same quantifiers before and after substitution.