Efficient Certified Resolution Proof Checking

Luís Cruz-Filipe[1], Joao Marques-Silva[2], and Peter Schneider-Kamp[1]

[1] Department of Mathematics and Computer Science,
University of Southern Denmark, Odense, Denmark
{lcf,petersk}@imada.sdu.dk
[2] LaSIGE, Faculty of Science, University of Lisbon, Lisbon, Portugal
jpms@ciencias.ulisboa.pt

Abstract. We present a novel propositional proof tracing format that eliminates complex processing, thus enabling efficient (formal) proof checking. The benefits of this format are demonstrated by implementing a proof checker in C, which outperforms a state-of-the-art checker by two orders of magnitude. We then formalize the theory underlying propositional proof checking in Coq, and extract a correct-by-construction proof checker for our format from the formalization. An empirical evaluation using 280 unsatisfiable instances from the 2015 and 2016 SAT competitions shows that this certified checker usually performs comparably to a state-of-the-art non-certified proof checker. Using this format, we formally verify the recent 200 TB proof of the Boolean Pythagorean Triples conjecture.

1 Introduction

The practical success of Boolean Satisfiability (SAT) solvers cannot be overstated. Generally accepted as a mostly academic curiosity until the early 1990s, SAT solvers are now used ubiquitously, in a variety of industrial settings, and with an ever increasing range of practical applications [6]. Several of these applications are safety-critical, and so in these cases it is essential that produced results have some guarantee of correctness [34].

One approach investigated over the years has been to develop formally derived SAT solvers [36,31,32,7]. These works all follow the same underlying idea: formally specify SAT solving techniques within a constructive theorem prover and apply program extraction (an implementation of the Curry–Howard correspondence) to obtain a certified SAT solver. Unfortunately, certified SAT solvers produced by this method cannot match the performance of carefully hand-optimized solvers, as these optimizations typically rely on low-level code whose correctness is extremely difficult to prove formally, and the performance gap is still quite significant.

An alternative approach that has become quite popular is to *check* the results produced by SAT solvers, thus adding some level of assurance regarding the computed results. This line of work can be traced at least to the seminal work of Blum and Kannan [8], with recent work also focusing on certifying algorithms and their verification [33,1]. Most SAT checkers expect the SAT solver to produce a witness

of its result, and then validate the witness against the input formula. For satisfiable instances, this is a trivial process that amounts to checking the computed satisfying assignment against the input formula. For unsatisfiable instances, since SAT is known to be in NP and believed not to be in coNP, it is unlikely that there exist succinct witnesses, in the worst case. As a result, the solution in practice has been to output a *trace* of the execution of the SAT solver, which essentially captures a resolution proof of the formula's unsatisfiability. Although this approach finds widespread use [16,44,35,26,25,5,37,20,41,19,42,21,42,18,24,22,43], and has been used to check large-scale resolution proofs [27,28,9,29,23], its main drawback is that there still is effectively no guarantee that the computed result is correct, since the proof checker has again not been proven correct.

Combining these two approaches, several authors [40,14,15,2,41,21] have experimented with the idea of developing *certified* proof checkers, i.e. programs that check traces of unsatisfiability proofs and that have themselves been formally proven correct. However, all these approaches are limited in their scalability, essentially for one of two reasons: (1) information about deletion of learned clauses is not available nor used [40,14,15,2]; and (2) the formats used to provide proof traces by SAT solvers still require the checker to perform complex checking steps [20,41,21,43], which are very difficult to optimize.

In this paper we examine the fundamental reasons for why these attempts do not scale in practice, and propose a resolution proof trace format that extends the one developed in recent work [20,19,42,21] by incorporating enough information to allow the reconstruction of the original resolution proof with minimum computational effort. This novel proof trace format impacts resolution proof checking in a number of fundamental aspects. First, we show how we can implement an (uncertified, optimized) proof checker in C whose run times are negligible when compared to those of state-of-the-art checkers, in particular drat-trim [42,17]. Second, we capitalize on the simplicity of the new proof format to formalize the proof verification algorithm inside the theorem prover Coq. Third, we extract a certified checker from this formalization and show that it performs comparably with drat-trim on a number of significant test cases. As a consequence, this certified checker is able to verify, in reasonable time, the currently largest available resolution proof, namely the 200 TB proof of the unsatisfiability of a SAT encoding of the Boolean Pythagorean Triples conjecture [23].

The paper is organized as follows. Section 2 briefly summarizes basic SAT and proof checking definitions, and presents a brief overview of the Coq theorem prover and its extraction mechanism. Section 3 provides an overview of the best known resolution proof formats proposed in the recent past. Section 4 introduces the novel resolution proof trace format and outlines the pseudo-code of a verification algorithm, which is then implemented in C. Section 4 also compares its performance to that of drat-trim [42]. Section 5 then describes a formalization of the SAT problem in Coq, which includes a specification of the pseudo-code in the previous section and a proof of its soundness. By applying the program extraction capabilities of Coq, we obtain a certified checker in OCaml, which we evaluate on the same test set as our uncertified C checker. Section 6 details

the performance of the certified checker on the verification of the proof of the Pythagorean Boolean Triples conjecture. The paper concludes in Section 7.

2 Preliminaries

Standard Boolean Satisfiability (SAT) definitions are assumed throughout [6]. Propositional variables are taken from a set X. In this work, we assume $X = \mathbb{N}^+$. A literal is either a variable or its negation. A clause is a disjunction of literals, also viewed as a set of literals. A conjunctive normal form (CNF) formula is a conjunction of clauses, also viewed as a set of clauses. Formulas are represented in calligraphic font, e.g. $\mathcal{F}$, with $\mathrm{var}(\mathcal{F})$ denoting the subset of X representing the variables occurring in $\mathcal{F}$. Clauses are represented with capital letters, e.g. C. Assignments are represented by a mapping $\mu : X \to \{0, 1\}$, and the semantics is defined inductively on the structure of propositional formulas, as usual. The paper focuses on CDCL SAT solvers [6]. The symbol $\vDash$ is used for entailment, whereas $\vdash_u$ is used for representing the result of running the well-known unit propagation algorithm.

This paper develops a formalized checker for proofs of unsatisfiability of propositional formulas using the theorem prover Coq [4]. Coq is a type-theoretical constructive interactive theorem prover based on the Calculus of Constructions (CoC) [10] using a propositions-as-types interpretation. Proofs of theorems are terms in the CoC, which are constructed interactively and type checked when the proof is completed; this final step ensures that the correctness of the results obtained in Coq only depends on the correctness of the type checker – a short piece of code that is much easier to verify by hand than the whole system.

A particular feature of Coq that we make use of in this paper is program extraction [30], which is an implementation of the Curry–Howard correspondence for CoC and several functional programming languages (in our case, OCaml). Programs thus obtained are correct-by-construction, as they are guaranteed to satisfy all the properties enforced by the Coq term they originate form. The CoC includes a special type `Prop` of propositions, which are understood to have no computational content; in particular, it is not allowed to define computational objects by case analysis on a term whose type lives in `Prop`. This allows these terms to be removed by program extraction, making the extracted code much smaller and more efficient; however, all properties of the program that they express are still valid, as stated by the soundness of the extraction mechanism.

3 Propositional Proof Trace Formats

The generation of resolution proof traces for checking the results of SAT solvers has been actively studied since the early 2000s [16,44]. Over the course of the years, different resolution proof tracing formats and extensions have been proposed [44,16,35,25,5,37,39,20,41,19,21,42,18,24,22,43]. These all boil down to listing information about the clauses learned by CDCL SAT solvers, with recent efforts allowing an extended set of operations [20,41]. Resolution proof traces can list the literals of each learned clause [37,39,19,21,42,43], the labels of the clauses used for learning each clause, or both [37,39,5]. Moreover, the checking of proof

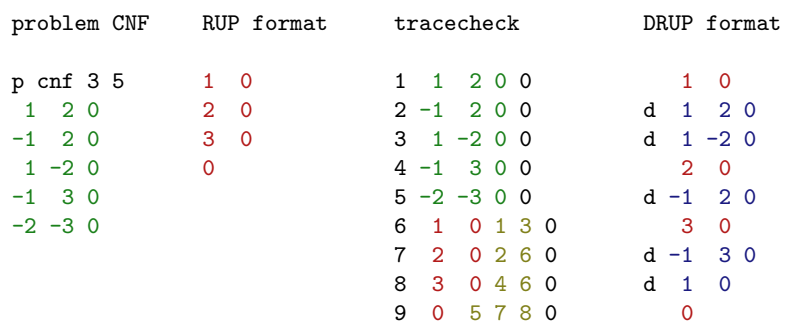

Fig. 1: Examples of trace formats (example adapted from [21]; with original clauses in green, deletion information in blue, learnt clauses in red, and unit propagation information in yellow)

traces can traverse the trace from the start to the end, i.e. *forward checking*, or from end to the start, i.e. *backward checking*. In addition, the checking of proof traces most often exploits one of two key mechanisms. One validation mechanism uses trivial resolution steps (TVR) [3]. This is a restriction over the already restricted input resolution [39]. For proof checking purposes it suffices to require that every two consecutively listed clauses *must* contain a literal and its complement (and obviously not be tautologous). Another validation mechanism exploits the so-called reverse unit propagation (RUP) property [16]. Let $\mathcal{F}$ be a CNF formula, and C be a clause learned from $\mathcal{F}$. Thus, we must have $\mathcal{F} \models C$. The RUP property observes that, since $\mathcal{F} \land \neg C \models \bot$, then it is also true that $\mathcal{F} \land \neg C \vdash \bot$. The significance of the RUP property is that proof checking can be reduced to validating a sequence of unit propagations that yield the empty clause. More recent work proposed RAT property checking [20,43]. The resulting format, DRAT, enables extended resolution proofs and, as a result, a wide range of preprocessing techniques [20,42,22].

A few additional properties of formats have important impact on the type of resulting proof checking. Some formats do not allow for clause deletion. This is the case with the RUP [37,39] and the *trace* [5] formats. For formats that generate clause dependencies, some will allow clauses *not* to be ordered, and so the checker is required to infer the correct order of steps.

Example 1. Figure 1 samples the proof tracing formats RUP, `trace`, and DRUP. (Compared to DRUP, the DRAT format is of interest when extended resolution is used. Every DRUP proof is by definition also a DRAT proof.) With the exception of the more verbose RES format, earlier formats did not allow for clause deletion. The DRUP format (and the more recent DRAT format) allow for clause deletion. A number of different traces would represent DRAT traces, including the DRUP trace shown.

Table 1 summarizes some of the best known formats, and their drawbacks. RES [37,39] is extremely verbose, separately encoding each resolution step, and is not in current use. RUP [37,39] and `trace` [5] do not consider clause deletion,

Table 1: Comparison of some of the best known proof tracing formats

Format	Clause Dependencies	Clause Literals	Clause Deletion	Clause Reordering	RAT Checking	Drawbacks
RES [37,39]	Yes	Yes	Yes	No	No	Size, RAT
RUP [37,39]	No	Yes	No	No	No	Deletion, RAT
`trace` [5]	Yes	Yes	No	Yes	No	Deletion, Reordering, RAT
DRUP [19,21]	No	Yes	Yes	Yes	No	RAT, Reordering
DRAT [42,43]	No	Yes	Yes	No	Yes	Complex checking

and so are inadequate for modern SAT solvers. DRUP addresses most of the drawbacks of earlier formats, and has been superseded by DRAT, which provides an extended range of operations besides clause learning.

A number of guidelines for implementing resolution proof trace checking have have emerged over the years. First, backward checking is usually preferred, since only the clauses in some unsatisfiable core need to be checked. Second, RUP is preferred over checking TVR steps [20,19,42,21,43], because the format becomes more flexible. Third, the SAT solver is often expected to minimize the time spent generating the proof trace. This means that, for formats that output clause dependencies, these are in general unordered. Moreover, modern checkers also carry out the validation of the RAT property [20,42,43]. These observations also indicate that recent work on checking of resolution proof traces has moved in the direction of more complex checking procedures.

Besides efficient checking of resolution proof traces, another important line of work has been to develop certified checkers. Different researchers exploited existing proof formats to develop certified proof checkers [40,14,15,2]. The main drawback of this earlier work is that it was based on proof formats that did not enable clause deletion. For large proofs, this can result in unwieldy memory requirements. Recent work addressed this issue by considering proof formats that enable clause deletion [20,41,43]. Nevertheless, this recent work builds on complex proof checking (see Table 1) and so does not scale well in practice.

Given past evidence, one can argue that, in order to develop efficient certified resolution proof checkers, proof checking *must* be as simple as possible. This has immediate consequences on the proof format used, and also on the algorithm used for checking that format. The next section details our proposed approach. The proposed format requires enough information to enable a checking algorithm that minimizes the processing effort. The actual checking algorithms exploits the best features of TVR and RUP, to enable what can be described as *restricted reverse unit propagation*.

4 Introducing the GRIT Format

As described in the section above, an important aspect in the design of propositional proof trace formats has been the desire to make it easy for SAT solvers

to produce a proof in that format. As a consequence, all the major proof trace formats have left some complex processing to the proof checker:

- The DRUP and DRAT formats specify the clauses learnt, but they do not specify the clauses that are used in reverse unit propagation to verify redundancy of these clauses. Thus, proof checkers need to implement a full unit-propagation algorithm.
- The *trace* format specifies which clauses are used in reverse unit propagation, but it deliberately leaves the order of these undetermined. Thus, proof checkers still need to implement a unit-propagation algorithm, though limited to the clauses specified.

Experience from recent work verifying large-scale proof [12,13], co-authored by two of the authors of this work, suggests that fully eliminating complex processing is a key ingredient in developing efficient proof checkers that scale to very large proofs. Furthermore, in the concrete case of unit-propagation, efficient algorithms rely on pointer structures that are not easily ported to the typical functional programming setting used in most theorem provers..

Based on these observations as well as on the importance of deleting clauses that are no longer needed [20,19], we propose a novel proof trace format that includes deletion and fully eliminates complex processing, effectively reducing unit-propagation to simple pre-determined set operations.

4.1 The Format

The *Generalized ResolutIon Trace (GRIT)* format builds on the *trace* format, but with two fundamental changes:

- We fix the order of the clauses dependencies given as a witness for each learnt clause to be one, in which unit propagation is able to produce the empty clause. This is a *restriction* of the freedom allowed by the *trace* format.
- In addition to the two types of lines specifying original and learnt clauses, we *extend* the format with a third type of line for deletions. These lines start with a `0` followed by a list of clause identifiers to delete and end with a `0`, and are thus easily distinguishable from the other two types of lines that start with a positive integer.

These changes are minimal w.r.t. achieving the integration of deletion and the elimination of complex processing, and in particular the new lines keep some of the properties that make the *trace* format easy to parse (two zeroes per line; the integers between those zeroes are clause identifiers). In this way, the changes follow the spirit of the extension of the RUP format to DRUP and later DRAT, just with *trace* as the point of departure.

Figure 2 shows how the GRIT version of our running example from Figure 1 incorporates the deletion information from the DRUP format into a *trace*-style proof, where the clause dependencies have been reordered to avoid the complexity of checking the RUP property by full unit propagation, instead facilitating the application of restricted reverse unit propagation.

The full syntax of the GRIT format is given by the grammar in Figure 3, where for the sake of sanity whitespace (tabs and spaces) is ignored. Here, addi-

```
problem CNF          tracecheck              DRUP format        GRIT format

p cnf 3 5            1  1  2 0 0            1  0               1  1  2 0 0
 1  2 0              2 -1  2 0 0            d  1  2 0          2 -1  2 0 0
-1  2 0              3  1 -2 0 0            d  1 -2 0          3  1 -2 0 0
 1 -2 0              4 -1  3 0 0               2  0            4 -1  3 0 0
-1  3 0              5 -2 -3 0 0            d -1  2 0          5 -2 -3 0 0
-2 -3 0              6  1  0 1 3 0             3  0            6  1  0 1 3 0
                     7  2  0 2 6 0          d -1  3 0          0  1  3 0
                     8  3  0 4 6 0          d  1  0            7  2  0 6 2 0
                     9  0  5 7 8 0             0               0  2  0
                                                               8  3  0 6 4 0
                                                               0  4  6 0
                                                               9  0  7 8 5 0
```

Fig. 2: Synthesis of the GRIT format (with original clauses in green, deletion information in blue, learnt clauses in red, and unit propagation information in yellow).

$$
\begin{array}{lcl}
\langle proof \rangle & = & \{\langle line \rangle\} \\
\langle line \rangle & = & (\langle original \rangle \mid \langle learnt \rangle \mid \langle delete \rangle), \text{"}\backslash n\text{"} \\
\langle original \rangle & = & \langle id \rangle, \langle clause \rangle, \text{"0"}, \text{"0"} \\
\langle learnt \rangle & = & \langle id \rangle, \langle clause \rangle, \text{"0"}, \langle idlist \rangle, \text{"0"} \\
\langle delete \rangle & = & \text{"0"}, \langle idlist \rangle, \text{"0"} \\
\langle clause \rangle & = & \{\langle lit \rangle\}, \text{"0"} \\
\langle idlist \rangle & = & \langle id \rangle, \{\langle id \rangle\} \\
\langle id \rangle & = & \langle pos \rangle \\
\langle lit \rangle & = & \langle pos \rangle \mid \langle neg \rangle \\
\langle pos \rangle & = & \text{"1"} \mid \text{"2"} \mid \ldots \\
\langle neg \rangle & = & \text{"-"}, \langle pos \rangle
\end{array}
$$

Fig. 3: EBNF grammar for the GRIT format.

tions with respect to the original *trace* format are given in green. In addition to the extension with delete information, there is a semantic restriction on the list of clause identifiers marked in red, namely that the clause dependencies represented are in the order as specified above. Existing parsers for the *trace* format should be easy to extend this syntax.

4.2 The Checker

To obtain an empirical evaluation of the potential of the GRIT format, we implemented a proof checking algorithm based on restricted reverse unit propagation in C. The source code is available from [11]. While the C code is quite optimized, the general algorithm follows the pseudo code given in Figure 4 as 25 lines of fully-functional Python (also available from [11]).

The set of instances we considered consists of the 280 instances from the 2015 and 2016 main and parallel tracks of the SAT competition that could be shown to be UNSAT within 5000 seconds using the 2016 competition version of lingeling. For each of these instances, the original CNF and proof trace are

```
def parse(line):
  ints = [int(s) for s in line.split()]
  i0 = ints.index(0)
  return ints[0], set(ints[1:i0]), ints[i0+1:-1]
def verify(file):
  cs = {}
  for id, c, ids in (parse(line) for line in file):
    if not id:      # delete clauses
      for id in ids:  del cs[id]
    elif not ids:  # add original clause
      cs[id] = c
    else:          # check & add learnt clause
      d = c.copy()
      for i in ids:
        e = cs[i]-d
        if e:
          d.add(-e.pop())  # propagate
          assert not e     # is unit?
        else:  # empty clause reached
          cs[id] = c
          if not c:  return "VERIFIED"
          break
  return "NOT VERIFIED"
import sys
print(verify(open(sys.argv[1])))
```

Fig. 4: Fully functional checker for the GRIT format written in Python.

trimmed and optimized using drat-trim in backward checking mode. This is a side-effect of using drat-trim to generate proof traces in the GRIT format, and was applied in the same way to generate DRAT files from the original RUP files in order to ensure a level playing field.

The C-checker successfully verifies all 280 GRIT files in just over 14 minutes (843.64 s), while drat-trim requires more than a day to solve the corresponding DRAT files (109214.08 s) using backward mode. Executing drat-trim in forward mode incurred a runtime overhead of 15 % on the total set of trimmed and optimized instances. As expected, the overhead was even bigger when working on the original CNFs and proof traces. The quantitative results are summarized in the plots of Figure 5, with details available from [11].

This two-orders-of-magnitude speedup demonstrates the potential of using a file format for propositional resolution proof checking by restricted reverse unit propagation. Note that we currently do *not* output the GRIT format directly, but require a modified version of drat-trim as a pre-processor[1] in order to determine

[1] The modified version essentially uses drat-trim's tracecheck output, interleaving it with deletion information. The modified source code is available from [11].

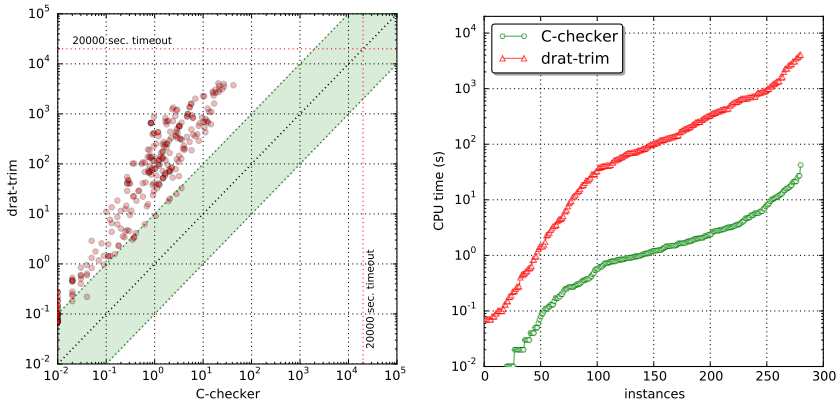

Fig. 5: Scatter and cactus plot comparing the runtime of the C-checker on GRIT files and drat-trim on the corresponding DRAT files.

both the order of clauses used in unit propagation, the set of original and learnt clauses relevant, and the deletion of clauses that are no longer needed.

While it is in principle thinkable to modify a SAT solver to output the GRIT format directly, building on [38], in this work our focus is on enabling sufficiently efficient certified proof checking. To this end, it seems fully acceptable to run an uncertified proof checker as a pre-processor to generate the oracle data enabling the application of restricted reverse unit propagation in a certified checker.

5 Coq Formalization

We now describe a Coq formalization that yields a certified checker of SAT proofs. We follow the strategy outlined in [12,13]: first, we formalize the necessary theoretical concepts (propositional satisfiability, entailment and soundness of unit propagation); then, we naively specify the verification algorithm; finally, we optimize this algorithm using standard computer science techniques to obtain feasible runtimes. In the interest of succintness, we only present the formalization obtained at the end of this process. The source files can be obtained from [11].

5.1 Formalizing propositional satisfiability

We identify propositional variables with Coq's binary natural numbers (type `positive`), and define a literal to be a signed variable. The type of literals is thus isomorphic to that of integers (excluding zero).

```
Inductive Literal : Type :=
  | pos : positive → Literal
  | neg : positive → Literal.
```

A clause is a set of literals, and a CNF is a set of clauses. For efficiency, there are two different definitions of each type, with mappings between them. A `Clause` is a `list Literal`, and is the type preferably used in proofs due to its simplicity; it is also the type used for inputting data from the oracle. A `CNF` is

a `BinaryTree Clause`, where the dependent type `BinaryTree` implements search trees over any type with a comparison operator. This is the type of the CNF given as input to the algorithm, which is built once, never changed, and repeatedly tested for membership. The working set uses two different representations of these types. A `SetClause` is a `BinaryTree Literal`, where in particular set differences can be computed much more efficiently than using `Clause`. Finally, an `ICNF` is a `Map {cl:SetClause | SC_wf cl}`, where `Map` is the Coq standard library's implementation of Patricia trees. The elements of an `ICNF` must be well-formed search trees (ensured by the condition in the definition of subset type); proofs of well-formedness do not contain computational meaning and are removed by extraction. In particular, every `SetClause` built from a `Clause` is well-formed.

A valuation is a function from positive numbers to Booleans. Satisfaction is defined for literals, clauses and CNFs either directly (as below) or by translating to the appropriate type (for `SetClause` and `ICNF`).

```
Definition Valuation := positive → bool.

Fixpoint L_satisfies (v:Valuation) (l:Literal) : Prop :=
  match l with
  | pos x ⇒ if (v x) then True else False
  | neg x ⇒ if (v x) then False else True
  end.

Fixpoint C_satisfies (v:Valuation) (c:Clause) : Prop :=
  match c with
  | nil ⇒ False
  | l ::  c' ⇒ (L_satisfies v l) ∨ (C_satisfies v c')
  end.

Definition C_unsat (c:Clause) : Prop := ∀ v:Valuation, ~(C_satisfies v c).

Fixpoint satisfies (v:Valuation) (c:CNF) : Prop :=
  match c with
  | nought ⇒ True
  | node cl c' c'' ⇒ (C_satisfies v cl) ∧ (satisfies v c') ∧ (satisfies v c'')
  end.

Definition unsat (c:CNF) : Prop := ∀ v:Valuation, ~(satisfies v c).

Definition entails (c:CNF) (c':Clause) : Prop :=
  ∀ v:Valuation, satisfies v c → C_satisfies v c'.
```

We then prove the intuitive semantics of satisfaction: a clause is satisfied if one of its literals is satisfied, and a CNF is satisfied if all its clauses are satisfied. Other properties that we need include: the empty clause is unsatisfiable; every non-empty clause is satisfiable; a subset of a satisfiable CNF is satisfiable; and a CNF that entails the empty clause is unsatisfiable.

```
Lemma C_satisfies_exist : ∀ (v:Valuation) (cl:Clause), C_satisfies v cl →
  ∃ l,  In l cl ∧ L_satisfies v l.
```

```
Lemma satisfies_remove : ∀ (c:CNF) (cl:Clause) (v:Valuation),
  satisfies v c → satisfies v (CNF_remove cl c).

Lemma unsat_subset : ∀ (c c':CNF),
  (∀  cl, CNF_in cl c → CNF_in cl c') → unsat c → unsat c'.

Lemma CNF_empty : ∀ c, entails c nil → unsat c.
```

5.2 Soundness of unit propagation

The key ingredient to verifying unsatisfiability proofs in GRIT format is being able to verify the original unit propagation steps. Soundness of unit propagation relies on the following results, formalizing the two relevant outcomes of resolving two clauses: a unit clause and the empty clause.

```
Lemma propagate_singleton : ∀ (cs:CNF) (c c':SetClause), ∀ l,
  entails cs (SetClause_to_Clause (SC_add (negate l) c')) →
  SC_diff c c' = (node l nought nought) → entails (CNF_add c cs) c'.

Lemma propagate_empty : ∀ (cs:CNF) (c c':SetClause),
  SC_diff c c' = nought → entails (BT_add Clause_compare c cs) c'.
```

We then define the propagation step: given an `ICNF`, a `SetClause` and a list of indices (of type `ad`, used in the implementation of `Map`), we successively check whether the set difference between the given clause and the clause with the first index from the list is empty (in which case we return `true`), a singleton (in which case we add the negation of the derived literal to the clause, remove the index from the list and recur), or a longer list of literals (and we return `false`). If the result is true, the initial clause is entailed by the original `ICNF`.

```
Fixpoint propagate (cs:ICNF) (c:SetClause) (is:list ad) : bool :=
  match is with
  | nil ⇒ false
  | (i:: is) ⇒ match SetClause_eq_nil_cons (SC_diff (get_ICNF cs i) c) with
    | inright _ ⇒ true
    | inleft H ⇒ let (l,Hl) := H in let (c',Hc) := Hl in
       match SetClause_eq_nil_cons c' with
       | inleft _ ⇒ false
       | inright _ ⇒ let (c'',_) := Hc in
          match SetClause_eq_nil_cons c'' with
          | inleft _ ⇒ false
          | inright _ ⇒ propagate cs (SC_add (negate l) c) is
  end end end end.

Lemma propagate_sound : ∀ (cs:ICNF) (c:SetClause) (is:list ad),
  propagate cs c is = true → entails cs c.
```

Observe that `propagate` is implementing a restricted version of reverse unit propagation, which in particular avoids complex processing for the next clause to use in unit propagation.

To check that a given formula is unsatisfiable, we start with an empty working set, and iteratively change it by applying actions given by the oracle. These can have three types: delete a clause; add a clause from the original CNF; or extend it with a clause that is derivable by unit propagation (and the oracle provides the clauses that should be used in this derivation).

```
Inductive Action : Type :=
  | D : list ad → Action
  | O : ad → Clause → Action
  | R : ad → Clause → list ad → Action.

Definition Answer := bool.

Function refute_work (w:ICNF) (c:CNF) (Hc:CNF_wf c) (O:Oracle)
  {measure Oracle_size O} : Answer :=
  match (force O) with
  | lnil ⇒ false
  | lcons (D nil) O' ⇒ refute_work w c Hc O'
  | lcons (D (i:: is)) O' ⇒ refute_work (del_ICNF i w) c Hc (lazy (lcons (D is) O'))
  | lcons (O i cl) O' ⇒ if (BT_in_dec _ _ _ __ cl c Hc)
                        then (refute_work (add_ICNF i cl' _ w) c Hc O') else false
  | lcons (R i nil is) O' ⇒ propagate w nought is
  | lcons (R i cl is) O' ⇒ andb (propagate w cl' is)
                                (refute_work (add_ICNF i cl' _ w) c Hc O')
  end.

Definition refute (c:list Clause) (O:Oracle) : Answer :=
  refute_work empty_ICNF (make_CNF c) (make_CNF_wf c) O.
```

Function `refute_work` implements this process over an oracle that is defined to be a lazy list. Lazy lists are defined exactly as lists, but with the second argument of `lcons` inside an invocation of an identity function. Likewise, `lazy` and `force` are defined as the identity. These additions are necessary to be able to extract a memory-efficient checker to OCaml. On extraction, these functions are mapped to the adequate OCaml constructs implementing laziness; although in principle this approach could break soundness of extraction, these constructs do indeed behave as identities. Without them, the extracted checker attempts to load the entire oracle data at the start of execution, and thus risks running out of memory for larger proofs.[2]

The initialization of auxiliary variables is done in `refute`, which only receives a CNF (in list form) and the oracle. This function then calls `refute_work` with an empty working set and the clause in binary tree representation. For legibility, we replaced proof obligations that are removed by extraction to an underscore, and wrote `cl'` for the result of converting `cl` to a `SetClause`.

The following result states soundness of `refute`: if `refute c O` returns `true`, then `c` is unsatisfiable. Since `O` is universally quantified, the result holds even if the oracle gives incorrect data. (Namely, because `refute` will output `false`.)

[2] Targeting a lazy language like Haskell would not require this workaround. However, in our context, using OCaml reduced computation times to around one-fourth.

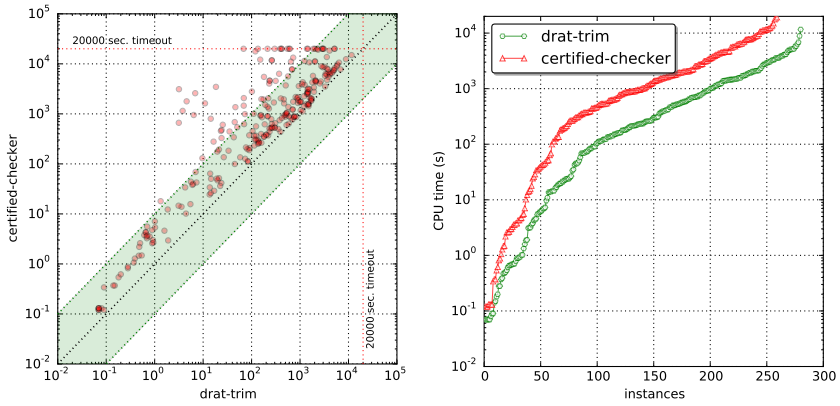

Fig. 6: Scatter and cactus plot comparing the runtime of our certified checker (including pre-processing) and drat-trim on the original proof traces from lingeling.

```
Theorem refute_correct : ∀ c 0, refute c 0 = true → unsat (make_CNF c).
```

5.3 Experimental Evaluation

In order to evaluate the efficiency of our formalized checker, we extracted it to OCaml. The extraction definition is available in the file `Extraction.v` from [11]. As is customary, we extract the Coq type `positive`, used for variable and clause identifiers, to OCaml's native integers, and the comparator function on this type to a straightforward implementation of comparison of two integers. This reduces not only the memory footprint of the verified checker, but also its runtime (as lookups in `ICNF`s require comparison of keys). It is routine to check that these functions are correct. Furthermore, as described above, we extract the type of lazy lists to OCaml's lazy lists.

We ran the certified extracted checker on the same 280 unsatisfiable instances as in the previous section, with a timeout of 20,000 seconds, resulting in 260 successful verifications and 20 timeouts. On the 260 examples, the certified checker runs in good 4 days and 18 hours (412469.50s) compared to good 2 days and 17 hours (234922.46s) required by the uncertified checker drat-trim. The pre-processing using our modified version of drat-trim adds another 2 days and 19 hours (241453.84s) for a total runtime of 7 days and good 13 (653923.34s). Thus, the extra degree of confidence provided by the certified checker comes at the price of approx. 2.8 times slower verification for these instances (180 % overhead).

The quantitative results on all 280 instances are summarized in the plots of Figure 6, where we added the pre-processing time to the time of the certified checker, with details available from [11].

The reason for the 20 timeouts can be found in the set implementation of our formalization. If one extracts the Coq sets to native Ocaml sets, there are no time-outs. We extracted such a version of the certified checker in order to check this hypothesis, as well as to assess the performance impact. And indeed, this version of our checker successfully verifies all 280 GRIT files in less time

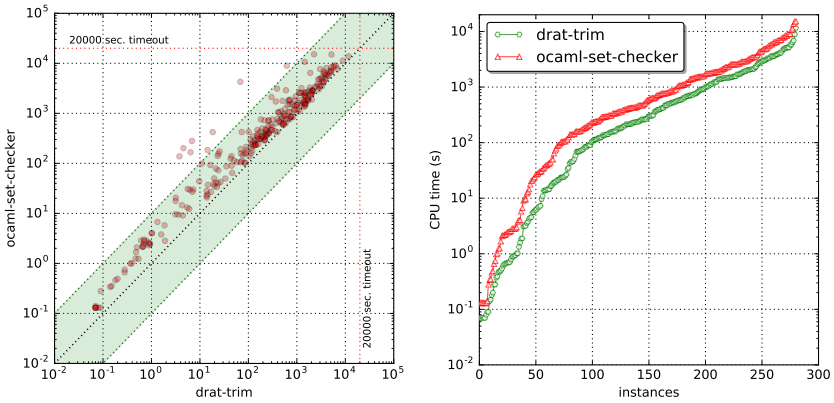

Fig. 7: Scatter and cactus plot comparing the runtime of a certified checker using Ocaml sets (including pre-processing) and drat-trim on the original proof traces from lingeling.

(186599.20s) than it takes to pre-process them using our modified drat-trim version (281516.13), and consequently the overhead of running a certified checker instead of an uncertified checker is down to 75 %. The quantitative results for this variant are summarized in the plots of Figure 7, with details available from [11].

6 Formally Veryifing the Boolean Pythagorean Triples proof

As a large-scale litmus test of our formally verified checker, we reconstituted the recent SAT-based proof of the Boolean Pythagorean Triples conjecture [23] (508 CPU days) using the incremental SAT solver *iGlucose*, transformed it into the GRIT format (871 CPU days) using our modified version of drat-trim, and formally verified that all 1,000,000 cases ("cubes") are indeed unsatisfiable (2608 days) using our certified checker (the original version, where all data structures except integers are extracted). This amounts to formally verifying the Boolean Pythagorean Triples conjecture (provided that its encoding as a propositional formula is correct).

The cactus plot in Figure 8 visualizes the distribution of runtime on the 1,000,000 cubes. The size of the reconstituted proof traces in RUP format was measured to be 175 TB. After transformation to the more detailed GRIT format, the proof traces filled a total 389 TB. During runtime, the maximum resident memory usage of the incremental SAT solver was 237 MB, while drat-trim in backward mode used up to 1.59 GB. Our certified checker reached a maximum of 67 MB of resident memory usage thanks to lazyness.

7 Conclusions & Research Directions

This paper revisits past work on proof checking aiming at the development of high-performance certified proof checkers. The paper proposes a new format,

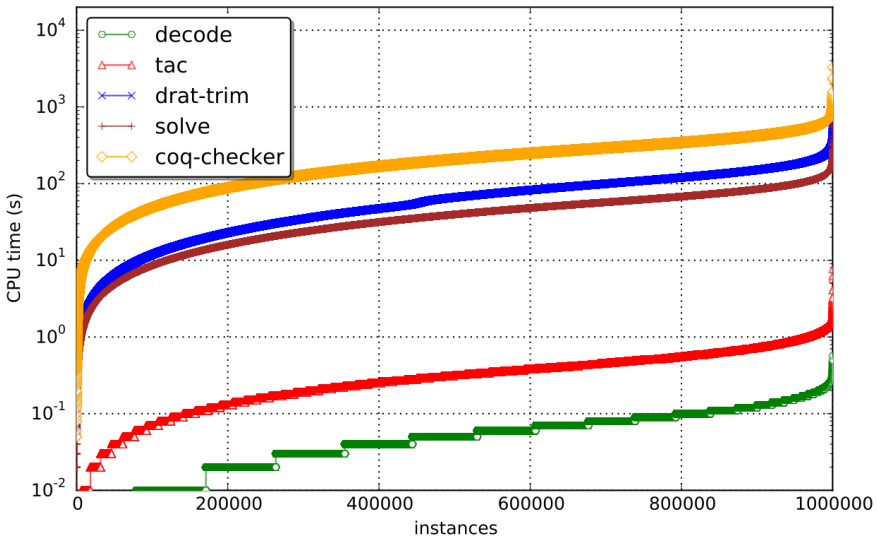

Fig. 8: Cactus plot comparing the runtimes for reconstituting the proof (decode and solve), transforming it into GRIT (drat-trim and tac), and formally verifying the GRIT files using our certified checker.

which enables a very simple proof checking algorithm. This simple algorithm is formalized in the Coq theorem prover from which an OCaml executable is then extracted.

The experimental results amply demonstrate the validity of the proposed approach. The C implementation of the checker is on average two orders of magnitude faster than what can be considered a reference C-implemented proof checker, drat-trim [42,17]. More importantly, the certified OCaml version of the checker performs comparably with drat-trim on problem instances from the SAT competitions. Perhaps more significantly, the certified checker has been used to formally verify the 200 TB proof of the Boolean Pythagorean Triples conjecture [23], in time comparable to the non-certified drat-trim checker.

Future work will address existing limitations of the approach. Currently, a modified version of drat-trim is used to generate the GRIT format. This can impact the overall running time, especially if the C-implemented checker for the GRIT format is to be used. This also includes modifying top performing SAT solvers to output the GRIT format, potentially based on A. Van Gelder's approach [38,39]. In addition, although not the focus on this paper, a natural extension of this work is to extend GRIT to be as general a format as DRAT, in particular by including support for the RAT property.

References

1. E. Alkassar, S. Böhme, K. Mehlhorn, and C. Rizkallah. A framework for the verification of certifying computations. *J. Autom. Reasoning*, 52(3):241–273, 2014.
2. M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In *CPP*, pages 135–150, 2011.
3. P. Beame, H. A. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res. (JAIR)*, 22:319–351, 2004.
4. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Texts in Theoretical Computer Science. Springer, 2004.
5. A. Biere. PicoSAT essentials. *JSAT*, 4(2-4):75–97, 2008.
6. A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.
7. J. C. Blanchette, M. Fleury, and C. Weidenbach. A verified SAT solver framework with learn, forget, restart, and incrementality. In *IJCAR*, pages 25–44, 2016.
8. M. Blum and S. Kannan. Designing programs that check their work. In *STOC*, pages 86–97, 1989.
9. R. L. Bras, C. P. Gomes, and B. Selman. On the Erdős discrepancy problem. In *CP*, pages 440–448, 2014.
10. T. Coquand and G. P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.
11. L. Cruz-Filipe and P. Schneider-Kamp. Grit format, formalization, and checkers. Available from: http://imada.sdu.dk/~petersk/grit/. Source codes also available from: https://github.com/peter-sk/grit.
12. L. Cruz-Filipe and P. Schneider-Kamp. Formalizing size-optimal sorting networks: Extracting a certified proof checker. In *ITP*, pages 154–169, 2015.
13. L. Cruz-Filipe and P. Schneider-Kamp. Optimizing a certified proof checker for a large-scale computer-generated proof. In *CICM*, pages 55–70, 2015.
14. A. Darbari, B. Fischer, and J. Marques-Silva. Formalizing a SAT proof checker in Coq. In *First Coq Workshop, published as technical report tum-i0919 of the Technical University of Munich*, 2009.
15. A. Darbari, B. Fischer, and J. Marques-Silva. Industrial-strength certified SAT solving through verified SAT proof checking. In *ICTAC*, pages 260–274, 2010.
16. E. I. Goldberg and Y. Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *DATE*, pages 10886–10891, 2003.
17. M. Heule. The DRAT format and DRAT-trim checker. CoRR, abs/1610.06229, 2016. Source code available from: https://github.com/marijnheule/drat-trim.
18. M. Heule and A. Biere. Proofs for satisfiability problems. In *All about Proofs, Proofs for All (APPA)*, July 2014. http://www.easychair.org/smart-program/VSL2014/APPA-index.html.
19. M. Heule, W. A. Hunt Jr., and N. Wetzler. Trimming while checking clausal proofs. In *FMCAD*, pages 181–188, 2013.
20. M. Heule, W. A. Hunt Jr., and N. Wetzler. Verifying refutations with extended resolution. In *CADE*, pages 345–359, 2013.
21. M. Heule, W. A. Hunt Jr., and N. Wetzler. Bridging the gap between easy generation and efficient verification of unsatisfiability proofs. *Softw. Test., Verif. Reliab.*, 24(8):593–607, 2014.
22. M. Heule, W. A. Hunt Jr., and N. Wetzler. Expressing symmetry breaking in DRAT proofs. In *CADE*, pages 591–606, 2015.

23. M. Heule, O. Kullmann, and V. W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In *SAT*, pages 228–245, 2016.
24. M. Heule, M. Seidl, and A. Biere. Efficient extraction of skolem functions from QRAT proofs. In *FMCAD*, pages 107–114, 2014.
25. T. Jussila, A. Biere, C. Sinz, D. Kröning, and C. M. Wintersteiger. A first step towards a unified proof checker for QBF. In *SAT*, pages 201–214, 2007.
26. T. Jussila, C. Sinz, and A. Biere. Extended resolution proofs for symbolic SAT solving with quantification. In *SAT*, pages 54–60, 2006.
27. B. Konev and A. Lisitsa. Computer-aided proof of Erdős discrepancy properties. *CoRR*, abs/1405.3097, 2014.
28. B. Konev and A. Lisitsa. A SAT attack on the Erdős discrepancy conjecture. In *SAT*, pages 219–226, 2014.
29. B. Konev and A. Lisitsa. Computer-aided proof of Erdős discrepancy properties. *Artif. Intell.*, 224:103–118, 2015.
30. P. Letouzey. Extraction in Coq: An overview. In *CiE 2008*, volume 5028 of *LNCS*, pages 359–369. Springer, 2008.
31. F. Maric. Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. *Theor. Comput. Sci.*, 411(50):4333–4356, 2010.
32. F. Maric and P. Janicic. Formalization of abstract state transition systems for SAT. *Logical Methods in Computer Science*, 7(3), 2011.
33. R. M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, 2011.
34. N. Shankar. Trust and automation in verification tools. In *ATVA*, pages 4–17, 2008.
35. C. Sinz and A. Biere. Extended resolution proofs for conjoining BDDs. In *CSR*, pages 600–611, 2006.
36. D. R. Smith and S. J. Westfold. Synthesis of satisfiability solvers. Technical report, Kestrel Institute, April 2008.
37. A. Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *ISAIM*, 2008.
38. A. Van Gelder. Improved conflict-clause minimization leads to improved propositional proof traces. In *SAT*, pages 141–146, 2009.
39. A. Van Gelder. Producing and verifying extremely large propositional refutations - have your cake and eat it too. *Ann. Math. Artif. Intell.*, 65(4):329–372, 2012.
40. T. Weber and H. Amjad. Efficiently checking propositional refutations in HOL theorem provers. *J. Applied Logic*, 7(1):26–40, 2009.
41. N. Wetzler, M. Heule, and W. A. Hunt Jr. Mechanical verification of SAT refutations with extended resolution. In *ITP*, pages 229–244, 2013.
42. N. Wetzler, M. Heule, and W. A. Hunt Jr. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *SAT*, pages 422–429, 2014.
43. N. D. Wetzler. *Efficient, mechanically-verified validation of satisfiability solvers*. PhD thesis, The University of Texas at Austin, 2015.
44. L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *DATE*, pages 10880–10885, 2003.