Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Formally Verifying the Solution to the Boolean Pythagorean Triples Problem

Luís Cruz-Filipe · Joao Marques-Silva · Peter Schneider-Kamp

Received: / Accepted:

Abstract The Boolean Pythagorean Triples problem asks: does there exist a binary coloring of the natural numbers such that every Pythagorean triple contains an element of each color? This problem was first solved in 2016, when Heule, Kullmann and Marek encoded a finite restriction of this problem as a propositional formula and showed its unsatisfiability. In this work we formalize their development in the theorem prover Coq. We state the Boolean Pythagorean Triples problem in Coq, define its encoding as a propositional formula and establish the relation between solutions to the problem and satisfying assignments to the formula. We verify Heule *et al.*'s proof by showing that the symmetry breaks they introduced to simplify the propositional formula are sound, and by implementing a correct-by-construction checker for proofs of unsatisfiability based on reverse unit propagation.

Keywords interactive theorem proving, large-scale proofs, SAT solving

Mathematics Subject Classification (2000) 68T15,68Q60,68N18,68Q17

Luís Cruz-Filipe
Department of Mathematics and Computer Science
University of Southern Denmark
Campusvej 55
5230 Odense M
E-mail: lcf@imada.sdu.dk

Joao Marques-Silva
LaSIGE, Faculty of Science, University of Lisbon, Lisbon, Portugal
E-mail: jpms@ciencias.ulisboa.pt

Peter Schneider-Kamp
Department of Mathematics and Computer Science
University of Southern Denmark
Campusvej 55
5230 Odense M
E-mail: petersk@imada.sdu.dk

1 Introduction

The Boolean Pythagorean Triples problem asks the following question: is it possible to partition the natural numbers into two sets such that no set contains a Pythagorean triple (three numbers a, b and c with $a^2 + b^2 = c^2$)? This problem is a particular instance of an important family of problems in Ramsey theory on the integers [22]: given an equation and an integer k, is there a coloring of the natural numbers using k colors such that there are no monochromatic solutions to the equation? If every k-coloring of the natural numbers admits a monochromatic solution, the equation is said to be *partition regular*. Several classical results in Ramsey theory establish partition regularity of particular equations, e.g. Schur's theorem, van der Waerden's theorem and Rado's theorem.

Partition regularity of the Pythagorean equation for $k = 2$ was finally established in 2016, when Heule, Kullmann and Marek [17] showed that it is already impossible to partition the set $\{1, \ldots, 7825\}$ into two sets such that none of them contains all elements of a Pythagorean triple. This proof was done by means of an encoding of this finite version of the problem into propositional logic (already used in [5]), which was then simplified and solved using the cube-and-conquer method [18].

More precisely, these authors considered the propositional formula

$$\bigwedge_{\substack{1 \leq a < b < c \leq 7825 \\ a^2 + b^2 = c^2}} (x_a \vee x_b \vee x_c) \wedge (\overline{x_a} \vee \overline{x_b} \vee \overline{x_c}) \tag{1}$$

where each x_i is a propositional variable and $\bar{\cdot}$ denotes logical negation. This formula exhaustively lists the sorted Pythagorean triples contained in $\{1, \ldots, 7825\}^3$ and requires that each triple contain at least one variable assigned to true and another assigned to false. A valuation satisfying the formula directly corresponds to a 2-coloring of the natural numbers up to 7825 without monochromatic Pythagorean triples.

The strategy of the proof is summarized in Figure 1. The propositional formula (1) was first simplified using blocked clause elimination and symmetry breaking. Afterwards, the problem was divided into one million *cubes*: a set of partial assignments that cover the whole space of possible valuations. Then, it was shown that (1) the conjunction of the simplified formula with any cube is unsatisfiable, and (2) the negation of the disjunction of all the cubes is unsatisfiable. As a consequence, the simplified formula (and therefore also the original formula) is unsatisfiable, which implies that the Boolean Pythagorean Triples problem has no solution.

Heule, Kullmann and Marek's proof relies heavily on SAT solvers that establish unsatisfiability of large propositional formulas. However, trusting that a formula is unsatisfiable simply because of the result of a SAT solver is not completely satisfactory, as there is no formal guarantee that the SAT solver is correct. For this reason, the same authors also produced a trace of their unsatisfiability proofs that they verified using an independently written checker, thereby checking their results independently. This improves confidence on the result, but it still requires trusting that the checker is correctly implemented (albeit a much weaker requirement).

The purpose of this work is to formalize Heule, Kullmann and Marek's proof that the Boolean Pythagorean Triples problem does not have a solution. To motivate the need for such a verification, we point out the steps in their original proof that depend on either informal arguments or trusting a computer program.

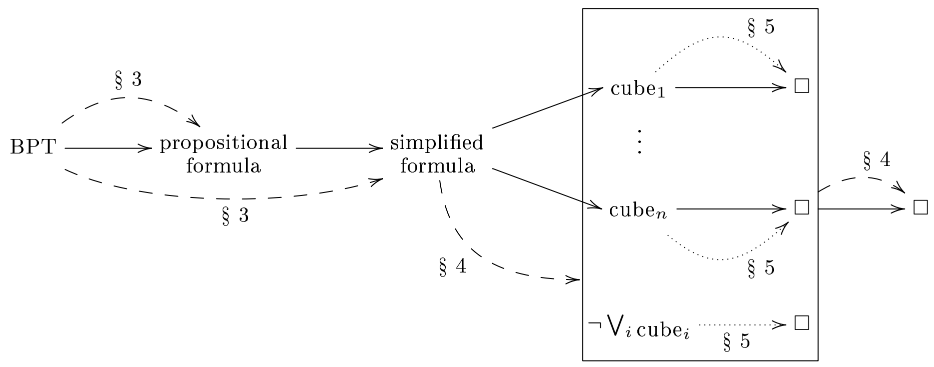

Fig. 1 The original proof and the different verification steps. The solid arrows denote the steps in the original proof [17]: a first propositional formula was generated by a C program, and subsequently simplified, divided and solved by SAT solvers. The dashed arrows denote steps that are directly formalized in Coq: the generation of propositional formulas that are proved to represent the original mathematical problem, directly and after simplification (Section 3); and the formal specification of the simple reasoning behind cube-and-conquer, as well as the generation of the formulas obtained by this methodology (Section 4). The dotted arrows denote steps that are verified by a certified checker extracted from a Coq formalization, namely the proofs of unsatisfiability of all the necessary propositional formulas (Section 5).

– The relationship between formula (1) and the Boolean Pythagorean Triples problem is stated informally, with an intuitive argument that it suffices to consider a finite approximation of the problem and that the propositional encoding of this approximation is correct.
– The actual instance of formula (1) that was used was generated using a C program whose soundness was never discussed.
– Although the simplification steps and the proofs of unsatisfiability were verified by an independent checker, this checker has in turn not been proven correct.
– Soundness of the cube-and-conquer methodology has not been proven formally.
– Applying cube-and-conquer requires manipulating formulas too large to process by hand, which have to be combined using e.g. command-line tools.

We do not claim that any of these issues presents a flaw in the original proof: the argument for encoding a finite subset of the problem (presented above) is simple; the connection between formula (1) and the original problem is simple to understand; the C program that generates the formula is simple enough to check for correctness manually; the likelihood that the handwritten checker accepts wrong traces is small;[1] soundness of cube-and-conquer is intuitively clear; and the file manipulation required in the last step consists only of copy and concatenation operations.

Still, a case for a fully formal proof of the Boolean Pythagorean Triples should avoid these pitfalls. In this paper, we undertake such an effort, formalizing the whole development in Figure 1 in the theorem prover Coq.

Contribution. We address all the issues discussed above, namely by:

– formalizing the Boolean Pythagorean Triples problem in Coq and showing that any solution yields a solution for all its finite approximations;

[1] This would require both the SAT solver and the checker to be flawed in a similar way, which is unlikely given that they were developed independently.

– generating formula (1) in Coq and showing that its satisfiability is equivalent to solvability of a particular finite instance of the Boolean Pythagorean Triples problem;
– formalizing the process of simplifying formula (1) in Coq, using a mathematical interpretation of the symmetry breaks applied;
– formalizing and extracting a correct-by-construction checker that is able to verify proofs of unsatisfiability based on reverse unit propagation (which suffices for all the formulas we need to consider);
– formalizing soundness of the cube-and-conquer methodology, extracting a correct-by-construction program that generates a set of formulas whose unsatisfiability implies unsatisfiability of formula (1).

We point out that the two last steps are generic and not in any way tailored to our particular goal of addressing the Boolean Pythagorean Triples problem. In other words, we obtain a general-purpose checker for proofs of unsatisfiability of propositional formulas based on reverse unit propagation, and a general-purpose tool for dividing a formula according to the cube-and-conquer methodology, given a set of cubes.

Heule *et al.*'s original development produced traces allowing all their proofs of unsatisfiability to be reproduced independently. These traces total around 200 TB of data, which led to their proof being described as "the largest mathematical proof ever". Our verification requires enriching their traces, making them approximately twice as large (in an uncompressed text format), and fully processing them, making it probably the largest formal verification at the time of writing. For comparison, recent large-scale formalizations relying on proof witnesses generated independently were able to process a few tens of GB of data [8,20].

We report experiments run on the Abacus 2.0 supercomputer of the DeIC National HPC Centre at the University of Southern Denmark. The nodes used were equipped with 64 GB RAM and 12 CPU cores (Intel(R) Xeon(R) CPU E5-2680 v3) able to run 24 threads in parallel.

Structure. Section 2 introduces the reader to the necessary background for reading this work: a contextualization of the Boolean Pythagorean Triples problem; an overview of the relevant SAT-solving techniques, in particular of reverse unit propagation and the cube-and-conquer method; and a bird's-eye view of the Coq theorem prover and its language.

Section 3 describes the formalization of the Boolean Pythagorean Triples problem in Coq and its encoding as a family of propositional formulas, together with a proof that unsatisfiability of any of these formulas implies that there is no solution to the Boolean Pythagorean Triples problem. We also discuss the symmetry breaks used in [17], implement them in Coq, and show their soundness.

In order to apply the divide-and-conquer methodology in the proof from [17], in Section 4 we formalize the algorithm behind the cube-and-conquer strategy [18] and show its soundness.

Section 5 focuses on the problem of efficiently validating proofs of unsatisfiability produced by an untrusted SAT solver. We formalize an algorithm for verifying reverse unit propagation in Coq, establish its soundness, and apply program extraction to obtain a certified checker that can verify a restricted class of proofs of unsatisfiability that suffices for our problem. Chaining the development in these three sections, we establish that there is no solution to the Boolean Pythagorean Triples problem using certified techniques.

We conclude in Section 6.

Publication history. These results were previously published in conference proceedings: the soundness of the encoding of the problem in SAT, of the symmetry breaks, and of the division strategy (cube-and-conquer) were described in [10], while verifying proofs of unsatisfiability based on reverse unit propagation in the general case was the topic of [9].

2 Background

2.1 The Coq theorem prover

Our formalization is developed using the interactive proof assistant Coq [2]. Coq is one successful member of a family of theorem provers based on dependent type theory: by means of a propositions-as-types interpretation, logic formulas are viewed as types, and the user interactively builds terms inhabiting those types. The same correspondence allows these terms to be viewed as proofs of the original proposition. The advantage of this family of theorem provers is that it is easy to check proofs automatically, once they have been built: type checking in these languages is decidable, and implementable by a small kernel that is simple enough to be formally checked for correctness. The remaining interface does not need to be trusted: if an error in the program allows for a wrong proof-term to be produced, the type checker will detect it. This property is usually described as saying that these theorem provers have a *small proof kernel.* (See e.g. [32] for an overview of the major theorem provers currently in use.)

The type theory underlying Coq is the Calculus of Constructions [6], a dependent type theory with inductive and co-inductive types. We highlight some of the features of the Calculus of Constructions that are most relevant to our work.

The logic corresponding to the Calculus of Constructions is intuitionistic (i.e., the principle of excluded middle $\varphi \vee \neg\varphi$ and the rule of double negation $\varphi \leftrightarrow \neg\neg\varphi$ do not hold in general). As a consequence, it is possible to implement a realizability interpretation in terms of program extraction [24], a mechanism by means of which proof terms are converted to programs in a suitable functional programming language (in our case, OCaml) whose correctness is guaranteed by their original type. Furthermore, the Calculus of Constructions includes a special type `Prop` whose elements are computationally irrelevant: they are used to express properties of data, and data cannot depend on them. Program extraction eliminates all terms whose type lives in `Prop`, thereby significantly reducing the size of the programs generated.

Although the general principle of excluded middle is not valid intuitionistically, there are a number of predicates for which it is possible to prove that they either hold or do not hold. Such predicates are called *decidable*; decidable predicates play an important role in our formalization, as they are extracted to programs that allow us to do case analysis on a given property. The Coq syntax for expressing that we can decide between `A` and `B` is `{A}+{B}`. For example, the formula `∀ (m n:nat),{n=m}+{n≠ m}` expresses that equality of two natural numbers is decidable, and it is extracted to a function that, given two numbers, returns `left` if they are equal and `right` if they are not. The actual proofs of (in)equality are not computed, as they have type `Prop`, but the soundness of program extraction guarantees that the semantics of this function is as described.

2.2 SAT solving in a nutshell

A *propositional formula* over a set of *propositional variables* is either one of these variables, a negation of a propositional formula, or the conjunction or disjunction of two propositional formulas. Propositional variables and their negations are referred to as *literals*. The *dual literal* $\bar{\ell}$ of a literal ℓ is defined as $\bar{x} = \neg x$ and $\overline{\neg x} = x$. An *assignment* or *valuation* is a mapping from propositional variables to the set of Boolean values $\{\mathsf{true}, \mathsf{false}\}$; assignments are extended homomorphically to propositional formulas, interpreting Boolean operations as usual. A *satisfying assignment* of a formula F is an assignment I such that $I(F) = \mathsf{true}$, which we denote as $I \models F$; in this case, we also say that I is a *model* of F. A formula F is *satisfiable* if it has a model, and *unsatisfiable* otherwise. A formula F entails a formula G, written $F \models G$, if all models I of F are also models of G.

The *Boolean satisfiability problem*, commonly abbreviated to SAT, asks whether a given propositional formula is satisfiable. This problem is known to be decidable and NP-complete [4]. The process of determining the answer to this problem is described as SAT solving, and programs that solve this problem are known as SAT solvers.

For the purpose of applying SAT solving algorithms, propositional formulas are written down in or converted to *conjunctive normal form* (CNF), i.e., a conjunction of disjunctions of literals. Formulas in CNF are interchangebly represented by sets of sets of literals, where the inner sets of literals, often referred to as *clauses*, are interpreted as disjunctions. The empty clause is usually denoted as $\Box$.

Solving SAT. The classical algorithm for SAT solving is *propositional resolution.* In this process, from clauses containing dual literals we are allowed to derive their *resolvent* clause, defined as their disjunction without the dual literals. Formally, given two clauses $C \vee \ell, C' \vee \neg\ell \in F$, their resolvent is $C \vee C'$. We define the *resolution relation* $\rightarrow_R$ on CNFs as $F \rightarrow_R F'$ if there are clauses C and C' and a literal ℓ such that $C \vee \ell, C' \vee \neg\ell \in F$ and $F' = F \wedge (C \vee C')$. Resolvents are entailed by the two resolved clauses, and consequently $F \rightarrow_R F'$ implies that $F \models F'$.

Most state-of-the-art SAT solvers are based on a conflict-driven clause-learning [29] variant of the DPLL algorithm [13], consisting of a dexterous combination of branching on variables, unit propagation, back-jumping, and clause learning, which we now briefly describe in sufficient detail to facilitate our proof checking.

The most basic step is branching on the value of a propositional variable, where the solver attempts to build a satisfying assignment for the formula by assigning one of the two possible truth values for the variable. This decision might have to be reconsidered later, as described below.

The next step is to use unit propagation to simplify the formula under the assumptions that were made. Unit propagation is performed by exhaustively applying a restricted form of propositional resolution, where one clause consists of exactly one literal. (Such clauses are called *unit clauses.*) We obtain the *unit propagation relation* $\rightarrow_U$ from $\rightarrow_R$ by restricting C' to be the empty clause $\Box$. The relation $\rightarrow_U$ is confluent and strongly normalizing, and we can thus define unit propagation on a set F as normalizing F w.r.t. $\rightarrow_U$, i.e., as $F\downarrow_U$.

If $F\downarrow_U$ contains the empty clause $\Box$, then F implies a contradiction, and thus F is unsatisfiable. In this case, previous branching decisions have to be reconsidered. This is achieved by constructing a *backjump clause* B that is entailed by F, and using this to determine which decisions to backtrack. By construction, the backjump clause

B satisfies $F \models B$, so the SAT solving algorithm can continue on $F \wedge B$, i.e., it has learned the clause B. If there are no branching decisions to reconsider, clause B is the empty clause $\square$ and the original CNF is unsatisfiable.

Checking the result. In practice, SAT solvers are extremely complex pieces software, typically implemented in low-level code using sophisticated low-level data structures to maximize efficiency. Thus, the assumption that the result of a SAT solver is correct is often deemed rather strong, motivating the need to check the result independently. If the result is positive, the SAT solver returns a satisfying assignment for the input formula, and it is trivial to check that this answer is correct.

Checking a negative answer derived from unit propagation is non-trivial, though: the only proof witnesses available are the clauses learned when encountering conflicts. Checking the proof then amounts to verifying that each learned B was indeed entailed by the corresponding formula F. This can be shown using a proof by contradiction strategy known as *reverse unit propagation*. Assume that there is a model I of F such that $I \not\models B$. Since B is a disjunctive clause, $\neg B$ is equivalent to a set of unit clauses, and $I \models \neg B$. If the empty clause $\square$ is reached by performing exhaustive unit propagation on $F \wedge \neg B$, i.e., if $\square \in (F \wedge \neg B) \downarrow_U$, then we have shown that $F \wedge \neg B$ is unsatisfiable and, consequently, $I \models B$, which contradicts the assumption that B is not entailed by F.

Although reverse unit propagation is not able to prove all valid entailments in propositional logic, it is strong enough to derive all the clauses that can be learned during conflict-driven clause-learning, making it thus a very useful tool when verifying the results provided by a SAT solver. A first generic format for representing such proofs was proposed in [16]. In this work, we use a Coq representation of the GRIT format [9], inspired by [12], which is described in Section 5.3. More expressive formats have very recently been proposed [7, 21], extending GRIT beyond the features needed for this work.

Cube-and-conquer. This methodology, introduced in [18], applies the principle of divide-and-conquer to SAT solving. The underlying idea is simple: instead of looking for a satisfying assignment for a particular formula, consider its conjunctions with different sets of literals (the cubes). The set of all cubes must be such that every possible assignment satisfies (at least) one cube. For example, let φ be a formula on two variables x and y, and consider the cubes $\{x\}$, $\{\bar{x}, y\}$ and $\{\bar{x}, \bar{y}\}$. Instead of trying to satisfy φ, we consider the three formulas $\varphi \wedge x$, $\varphi \wedge \bar{x} \wedge y$ and $\varphi \wedge \bar{x} \wedge \bar{y}$. If one of these is satisfiable, then we proved that φ is satisfiable; if all are unsatisfiable, then to establish unsatisfiability of φ we also need to check that the formula $\bar{x} \wedge (x \vee \bar{y}) \wedge (x \vee y)$ is unsatisfiable.

Lemma 1 (Soundness of cube-and-conquer) *Let φ be a CNF and $C = \{c_i\}_{i=1}^n$ be a set of sets of literals such that $\bigvee_{i=1}^n \left(\bigwedge c_i\right)$ is a tautology. Then φ is satisfiable iff there exists $1 \leq i \leq n$ such that $\varphi \wedge \bigwedge c_i$ is satisfiable.*

The proof of this lemma is omitted, as its formalization in Coq is described later on (namely, it is lemma `CubeAndConquer_lemma` on page 18).

In the cube-and-conquer methodology, the biggest challenge is finding the "right" set of cubes – namely, a set that makes the resulting (un)satisfiability proofs easy, but that does not generate too many subproblems. This process typically requires state-of-the art techniques, with complex heuristics and look-ahead strategies [25]. When checking proofs, however, this complex step is avoided – the cubes are given as part of the input.

3 Formalizing a Propositional Encoding

In this section, we describe the formalization of the Boolean Pythagorean Triples problem in Coq and its encoding as a family of propositional formulas parameterized on a natural number n. We first develop this encoding in the simplest possible way, and then apply two symmetry breaks that make it smaller. The encoding is proved correct before and after the symmetry breaks, in the sense that the propositional formulas obtained are satisfiable for all n if and only if the Boolean Pythagorean Triples problem has a solution. This process corresponds to the two leftmost dashed arrows in Figure 1.

This development depends on an encoding of propositional logic in Coq, which is the topic of the first subsection. We explain the relevant aspects of the Coq syntax in the presentation.

3.1 Propositional logic and satisfiability

We formalize a simple theory of propositional logic tailored to the final objective of this work: verifying a proof of unsatisfiability of a particular formula, produced by a SAT solver. This impacts our formalization in two ways: first, we only capture the fragment of conjunctive normal forms, which is the fragment that SAT solvers act upon; second, some of our design choices reflect common practice in SAT solving – namely, using positive integers for atomic formulas and assigning numbers to clauses.

We identify propositional variables with Coq's binary natural numbers (type `positive`), and define a literal to be a signed variable.

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

The type of literals thus obtained is isomorphic to that of integers (excluding zero).

From literals we build clauses and CNFs: a clause is a set of literals, and a CNF is a list of clauses.

```
Definition Clause := list Literal.

Definition CNF := list Clause.
```

Valuations are simply functions from positive numbers to Booleans. Satisfaction is defined for `Literal`s, `Clause`s and `CNF`s in the expected way. All these functions are defined by pattern matching (performed by the Coq construct `match`). In `L_satisfies`, the literal argument is matched to either `pos x` or `neg x` (where `pos` and `neg` are the constructors of the inductive type `Literal`); valuation `v` satisfies literal `pos x` if `(v x)` is `true`, and it satisfies literal `neg x` if `(v x)` is `false`.[2] Satisfaction of either `Clause`s (`C_satisfies`) or `CNF`s (`satisfies`) are defined recursively on these terms, which are lists with constructors `nil` and `cons` (inlined as ::).

[2] The constants `true` and `false`, inhabiting the type `bool`, are distinguished in Coq from the singleton type `True` and the empty type `False`, which are logical propositions inhabiting the sort `Prop`.

```
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.

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

We define two important notions related to satisfaction: unsatisfiability (of a `CNF`) and entailment (of a `Clause` from a `CNF`). Negation is written in Coq as ~ (where `~A` is an abbreviation of the type `A` → `False` of functions from `A` to the empty type).

```
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 and only if one of its literals is satisfied, and a CNF is satisfied if and only if all its clauses are satisfied. It is convenient to express these equivalences as implications, as the corresponding results are then more easily applicable in Coq. In particular, we avoid the need for existential quantification in lemma `exist_C_satisfies` below.

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

Lemma exist_C_satisfies : ∀ (v:Valuation) (cl:Clause) (l:Literal),
  In l cl → L_satisfies v l → C_satisfies v cl.

Lemma satisfies_for_all : ∀ (v:Valuation) (c:CNF), satisfies v c →
  ∀ (cl:Clause), In cl c → C_satisfies v cl.

Lemma for_all_satisfies : ∀ (v:Valuation) (c:CNF),
  (∀ (cl:Clause), In cl c → C_satisfies v cl) → satisfies v c.
```

Other useful properties that we prove include: the empty clause is unsatisfiable; a subset of a satisfiable CNF is satisfiable; and a CNF that entails the empty clause is unsatisfiable.

```
Lemma C_unsat_empty : C_unsat nil.

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

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

3.2 The Boolean Pythagorean Triples problem in Coq

In this work, we use the binary representation of natural numbers as implemented by the Coq type `positive`. We define a (binary) coloring to be a function from `positive` to the type `bool` of Booleans. In other words, the two colors we use are `true` and `false`; this simplifies our subsequent development, as these are also the truth values that we use for propositional formulas. Indeed, the type `coloring` is equal (in Coq's type theory) to the type `Valuation`.

```
Definition coloring := positive → bool.
```

We identify Pythagorean triples by means of a predicate over triples of natural numbers, defined in the natural way.[3] This definition uses the overloading feature of arithmetical operators in Coq. The token `%positive` tells the Coq parser that the preceding expression should be interpreted over the type `positive`.

```
Definition pythagorean (a b c:positive) := (a*a + b*b = c*c)%positive.
```

Given three natural numbers, we can always decide whether they form a Pythagorean triple, since equality on the natural numbers is decidable. Decidability is a key ingredient to our work: a term of type `{A}+{B}`, where `A` and `B` are propositions, proves that one of `A` and `B` always holds, and that we can compute which one. A proof of this term is an algorithm to decide which of `A` and `B` is true, and Coq's extraction mechanism allows us to transform this proof into a functional program that implements this algorithm.

```
Lemma pythagorean_dec : ∀ (a b c:positive), {pythagorean a b c} + {~pythagorean a b c}.
```

Coq also allows an if-then-else syntax to define functions by case analysis on decidable predicates (motivated by the fact that we are typically interested in the case where `B` is `~A`). We use this feature later in this development.

The formula (1) ranges only over sorted Pythagorean triples. In order to show that this is enough, we need some simple properties of Pythagorean triples that we prove at this stage: if (a, b, c) is a Pythagorean triple, then $a < c$, $b < c$, $a \neq b$, and (b, a, c) is also a Pythagorean triple.

```
Lemma pythagorean_lt_1 : ∀ a b c, pythagorean a b c → (a < c)%positive.

Lemma pythagorean_lt_2 : ∀ a b c, pythagorean a b c → (b < c)%positive.

Lemma pythagorean_neq : ∀ a b c, pythagorean a b c → a ≠ b.

Lemma pythagorean_comm : ∀ a b c, pythagorean a b c → pythagorean b a c.
```

Finally, we formalize partition regularity (for $k = 2$) of the Pythagorean equation. We say that a coloring has the Pythagorean property (`pythagorean_prop`) if every Pythagorean triple contains two elements of different colors.

```
Definition pythagorean_prop (C:coloring) :=
  ∀ (a b c:positive), pythagorean a b c → (C a ≠ C b) ∨ (C a ≠ C c) ∨ (C b ≠ C c).
```

[3] Expressing this property as a predicate, rather than as a record type, simplifies our development.

3.3 The direct encoding

The next step is to construct a family of propositional formulas, parameterized on a natural number n, of which formula (1) is the particular case with $n = 7826$. To obtain these formulas, we first define a function that takes a unary natural number (of type `nat`) as argument and returns a list of all sorted Pythagorean triples with elements in $\{1, \ldots, n\}^3$ by double iteration. (The usage of unary numbers as recursive arguments simplifies recursive definitions.)

We implement the double recursion by means of two auxiliary functions. Function `inner_cycle` assumes `n` and `b` (the bound) to be fixed, and iterates through `m`. At each step, it computes the integer square root `sqrt` of `n*n+mN*mN` (where `mN` is obtained by converting `m` to a binary integer, using the mapping `Pos.of_nat`), checks whether these three values form a Pythagorean triple (using the predicate `pythagorean_dec`), and adds the triple (`mN`,`n`,`sqrt`) to the result if `sqrt<b`. The result is a list of all Pythagorean triples where the first element is always less than or equal to `m`, the second element is always `n`, and the third element is smaller than `b`. Since this function will be called with the same values for `n` and `m`, the triples we obtain are always ordered ascendingly.

```
Definition target_list := list (list positive).

Fixpoint inner_cycle (n:positive) (m:nat) (b:positive) : target_list :=
  match m with
  | 0 ⇒ nil
  | S m' ⇒ let mN := Pos.of_nat m in let sqrt := (Pos.sqrt (n*n+mN*mN)) in
           if (sqrt<?b)%positive
           then if (pythagorean_dec n mN sqrt)
                then (mN:: n:: sqrt:: nil) :: inner_cycle n m' b
                else (inner_cycle n m' b)
           else (inner_cycle n m' b)
  end.
```

The second loop is implemented in function `outer_cycle`, which recurs on `n`, calling `inner_cycle` with `n` as first and second argument and passing `b` unchanged. The result is a list containing all Pythagorean triples where: the first two elements are smaller than or equal to `n`, the third element is smaller than `b`, and the elements are in ascending order. List concatenation is written inline in Coq as `++` .

```
Fixpoint outer_cycle (n:nat) (b:positive) : target_list :=
  match n with
  | 0 ⇒ nil
  | S m ⇒ (inner_cycle (Pos.of_nat n) n b) ++ (outer_cycle m b)
  end.
```

Finally, by instantiating `b` with `n`, we obtain the list of all ordered Pythagorean triples whose elements are all smaller than `n`.

```
Definition BPT_list (n:nat) := outer_cycle n (Pos.of_nat n).
```

We could have defined `BPT_list` differently, for example using an accumulator. Our implementation has the advantage of having a very simple recursive structure that makes it very easy to reason about.

The next step is to transform the list of triples generated into propositional formulas, i.e. into elements of the type `CNF`.

We start by mapping `BPT_list` into a list of `Clause` by generating two `Clause`s from each triple (one with positive literals, another with negative literals, obtained by mapping the appropriate constructors of type `Literal` to each tuple). For $n = 7826$, this list contains exactly the clauses in formula (1).

```
Fixpoint target_formula (t:target_list) : CNF :=
  match t with
  | nil ⇒ nil
  | tuple::t' ⇒ (map pos tuple) :: (map neg tuple) :: (target_formula t')
  end.
```

```
Definition BPT_formula (n:nat) := target_formula (BPT_list n).
```

This formula corresponds, in our formalization, to a generalization of formula (1), where the bound 7825 has been replaced by $n - 1$. We now formally show that every solution to the Boolean Pythagorean Triples problem is also a valuation that makes all formulas in this family true. Since the types `coloring` and `Valuation` are actually the same in our formalization, we can prove that each adequate coloring *is* a satisfying valuation. In other words, if `C` is a coloring of the natural numbers such that no Pythagorean triple is monochromatic, then (`BPT_formula n`) is satisfiable for each `n`.

```
Lemma BPT_formula_sat : ∀ (C:coloring), pythagorean_prop C →
  ∀ (n:nat), satisfies C (BPT_formula n).
```

The proof of this result is in several steps. First we prove that every triple in `BPT_list n` contains two (distinct) elements `a` and `b` such that `C a` $\neq$ `C b`. This is established by induction over the definition of `BPT_list n`. From this property, a simple induction over the definition of `target_formula` establishes that every clause in `BPT_formula n` is satisfied by `C`.

The converse result also holds: if there is a coloring `C` that satisfies (`BPT_formula n`) for every `n`, then `C` establishes partition regularity of the Pythagorean equation for $k = 2$.

```
Lemma BPT_formula_sat' : ∀ (C:coloring), (∀ n, satisfies C (BPT_formula n)) →
  pythagorean_prop C.
```

The proof of this result starts by showing that, if (a, b, c) is a Pythagorean triple, then (`BPT_formula n`) contains either the two clauses $(x_a \vee x_b \vee x_c)$ and $(\overline{x_a} \vee \overline{x_b} \vee \overline{x_c})$ or the two clauses $(x_b \vee x_a \vee x_c)$ and $(\overline{x_b} \vee \overline{x_a} \vee \overline{x_c})$ for any $n > c$. (In the interest of legibility, we write these clauses in mathematical notation, rather than the corresponding Coq terms.)

In particular, if for some value of n the corresponding formula is unsatisfiable, then the Boolean Pythagorean Triples problem does not have a solution. Dually, if this formula is satisfiable, then there is a coloring of the natural numbers such that all Pythagorean triples not containing numbers larger than n are monochromatic. The predicate `pyth_lim_prop` is a restricted version of `pythagorean_prop`, ranging only over numbers smaller than its first argument (in this case, `TheN`).

```
Parameter TheN : nat.
```

```
Definition The_CNF := BPT_formula TheN.
```

```
Theorem Pythagorean_Theorem : unsat The_CNF → ∀ (C:coloring), ~pythagorean_prop C.
```

```
Theorem Pythagorean_dual_Theorem : ∀ C, satisfies C The_CNF → pyth_lim_prop TheN C.
```

In order to generate formula (1), we need to instantiate `TheN` in `The_CNF` to 7826. This can be done inside Coq (as long as it is invoked with adequate command line parameters to allow for the necessary resources); however, as we discuss below, later steps require applying the program extraction mechanism of Coq [24] to obtain a correct-by-construction OCaml program that is run independently. Therefore, we instead declare this variable as a `Parameter`. Technically, `Parameter`s are axioms, which can introduce inconsistencies in the formalization; however, since the type of `TheN` is `nat` (which is known to be inhabited), this particular parameter declaration preserves soundness. On the formalization level, we are quantifying universally over the natural numbers; on the computational level, we are delegating the instantiation of `TheN` to the extraction step.

As an experiment, we extracted `The_CNF`, set `TheN`, evaluated the corresponding expression in OCaml, and verified that we obtained formula (1). The representation is not exactly the same as that used in [17]: on the one hand, the variable those authors match to natural number n is $x_{n+10,000}$, rather than x_n; on the other hand, the indices assigned to the individual clauses are different. Otherwise, the two formulas coincide.

By instantiating `TheN` to 7825 instead, we obtain a smaller formula that is satisfiable. Our extracted program can in principle verify this, if given an adequate valuation as argument. Such a valuation can however only be computed by existing SAT solvers for a simplified version of this formula (see [17]).

3.4 The simplification process

The second step in the work described in [17] was simplifying formula (1) in order to make the SAT solving process simpler. Their simplification consisted of two steps: blocked claused elimination and symmetry breaking.

Blocked clause elimination [19] is a powerful technique that detects clauses in a CNF that can be removed, producing equisatisfiable formulas. Preservation of satisfiability is straightforward, since a valuation satisfying a CNF always satisfies all of its subsets; and this property actually suffices to show that the Boolean Pythagorean Problems has no solution. However, [17] discusses a stronger result: there is actually a coloring of the first 7824 natural numbers that includes no monochromatic Pythagorean triples. Verifying this claim with our formalization requires showing that these simplification techniques also preserve *un*satisfiability.

There is another reason for formalizing the simplification process in more detail: there is a simple mathematical argument that precisely describes what blocked clause elimination is doing in this particular situation.[4]

Lemma 2 *Let L be a set of Pythagorean triples and $(a, b, c) \in L$ be such that a does not occur in any other triple in L. If there is a binary coloring of the natural numbers such that no element of $L \setminus \{(a, b, c)\}$ is monochromatic, then there is also a binary coloring such that no element of L is monochromatic.*

Proof Let C be a binary coloring of the natural numbers such that no element of $L \setminus \{(a, b, c)\}$ is monochromatic. If (a, b, c) is not monochromatic under C, then C is the desired coloring. Otherwise, let C' be the coloring obtained from C by changing the color of a. Since a does not occur in any other triple in L, C' also does not make any element of $L \setminus \{(a, b, c)\}$ monochromatic, and by construction it does not make (a, b, c) monochromatic; thus, C' is the desired coloring.

[4] This simple argument was communicated informally by Marijn Heule. To the best of our knowledge, it has not been published elsewhere.

Of course, the result still holds if b or c, rather than a, do not occur in any other triple in L. Furthermore, it is not relevant that L is a list of triples: we can formalize a more general property.

Lemma 3 *Let t be a list of tuples of natural numbers, a be a number that does not occur in any element of t, ℓ be a tuple containing a and at least one other element, and C be a binary coloring such that no element of t is monochromatic under C. Then there exists a binary coloring C' such that $t \cup \{\ell\}$ is monochromatic under C'.*

In our development, t is simply an element of the type `target_list` defined earlier. We omit a mathematical proof of this lemma, as we now proceed to formalize it in Coq.

To write Lemma 3, we need to be able to count how many times a number occurs in the elements of a list of tuples. Rather than defining a general counting predicate, we inductively define two relations `no_occurrence` and `one_occurrence` for the two particular cases we need to consider. Having these specialized predicates, once again, simplifies our development.

```
Fixpoint no_occurrence (p:positive) (t:target_list) :=
  match t with
  | nil ⇒ True
  | (l:: t') ⇒ ~In p l ∧ no_occurrence p t'
  end.

Lemma no_occurrence_char : ∀ (p:positive) (t:target_list),
  no_occurrence p t ↔ ∀ l, In l t → ~In p l.

Fixpoint one_occurrence (p:positive) (t:target_list) :=
  match t with
  | nil ⇒ False
  | (l:: t') ⇒ (In p l ∧ no_occurrence p t') ∨ (~In p l ∧ one_occurrence p t')
  end.

Lemma one_occurrence_find : ∀ (p:positive) (t:target_list), one_occurrence p t →
  ∃ (l:list positive), In p l ∧ In l t ∧ (∀ l', In l' t → l ≠ l' → ~In p l').
```

The two characterization lemmas included restate the recursive definitions as global properties of the list `t`: `no_occurrence p t` holds if and only if `p` does not occur in any element of `t`, and `one_occurrence p t` holds if and only if `p` occurs in exactly one element of `t`. From `one_occurence_find` we can also prove that, if the particular tuple containing `p` is removed, then `p` does not occur in the result.

We define another predicate `colorful` characterizing colorings that do not make any element of a `target_list` monochromatic. We can now formally state Lemma 3 as lemma `colorful_add` below.

```
Definition colorful (C:coloring) (t:target_list) := ∀ (tuple:list positive),
  In tuple t → ∃ (a b:positive), In a tuple ∧ In b tuple ∧ C a ≠ C b.

Lemma colorful_add : ∀ (t:target_list) (a:positive), no_occurrence a t →
  ∀ (C:coloring), colorful C t → ∀ (b:positive), a ≠ b →
  ∀ (l:list positive), In a l → In b l → ∃ (C':coloring), colorful C' (l::t).
```

The proof of this lemma follows the informal proof above. We do case analysis on the possible values of (`C a`) and (`C b`). If they are distinct, then `C` is the desired coloring; otherwise, we flip the value of (`C a`) in `C'`, and use the fact that `a` does not occur in any other tuple to show that `C'` is `colorful` with respect to (`l:: t`).

In order to simplify formula (1), we iterate this argument starting from the set of all ordered Pythagorean triples in $\{1,\ldots,7825\}^3$. This allows removal of 2136 of the original 9472 triples. We formalize this in a robust way: given a list of numbers and a `target_list`, we iteratively consider each number in turn. If it occurs in exactly one tuple in the list, we remove that tuple and continue; otherwise, we leave the list unchanged and continue. This guarantees that Lemma 3 always applies to the final result, when we start from a list of Pythagorean triples. Recall that the if-then-else notation in Coq allows us to define functions by analysing which branch of a decidable disjunction holds. We use this notation in these definitions, making choices based on whether a list `l` contains an element `p` (`In_dec Pos.eq_dec p l`, where `Pos.eq_dec` is a term used for deciding equality of elements of type `positive`) or whether `one_occurrence p t` holds (which is decidable according to `one_occurrence_dec`).

```
Fixpoint remove_number (p:positive) (t:target_list) :=
  match t with
  | nil ⇒ nil
  | l:: t' ⇒ if (In_dec Pos.eq_dec p l) then remove_number p t'
                                       else l:: remove_number p t'
  end.

Fixpoint simplify (t:target_list) (l:list positive) :=
  match l with
  | nil ⇒ t
  | p:: l' ⇒ if (one_occurrence_dec p t) then simplify (remove_number p t) l'
           else simplify t l'
  end.

Definition ok_list (t:target_list) := ∀ (tuple:list positive),
  In tuple t → ∃ a b c, tuple = (a::b:: c:: nil) ∧ pythagorean a b c.

Lemma colorful_simplify : ∀ (t:target_list), ok_list t →
  ∀ (l:list positive) (C:coloring), colorful C (simplify t l) →
  ∃ (C':coloring), colorful C' t.
```

Lemma `colorful_simplify` is proved by induction using lemma `colorful_add`.

Finally, we apply this construction to the family of formulas `BPT_formula` constructed earlier, obtaining a family of simplified formulas depending not only on the original parameter n, but also on the list of numbers to be used for removal of tuples.

```
Definition simplified_BPT_formula (n:nat) (l:list positive) :=
  target_formula (simplify (BPT_list n) l).

Parameter The_List : list positive.

Definition The_Simple_CNF := simplified_BPT_formula TheN The_List.
```

As before, adding `The_List` as a `Parameter` does not compromise the soundness of our development, since `list positive` is trivially inhabited. There are two reasons for including it, rather than defining the iteration procedure from [17]: first, it simplifies the formalization, eliminating the need to formalize that procedure and prove its soundness; second, it significantly reduces computation time, since we do not need to analyze the formula in order to find out which element to remove next. The actual instance of `The_List` is later obtained from the trace of the original simplification proof in [17].

Specializing the results on the simplification procedure to these definitions, we prove that simplification preserves unsatisfiability. The converse implication is straightforward, since every clause in the simplified formula is present in the original one. Using this result, we can state and prove variants of `Pythagorean_Theorem` and `Pythagorean_dual_Theorem` for this simplified formula. The latter theorem is now existentially quantified, as we have no guarantee that the valuation satisfying `The_Simple_CNF` coincides with the adequate coloring of the relevant subset of natural numbers.

```
Theorem simplification_ok : unsat The_CNF ↔ unsat The_Simple_CNF.

Theorem Pythagorean_Theorem' :
  unsat The_Simple_CNF → ∀ (C:coloring), ~pythagorean_prop C.

Theorem Pythagorean_dual_Theorem' : ∀ C, satisfies C The_Simple_CNF →
  ∃ C', pyth_lim_prop TheN C'.
```

The second step in the simplification phase in [17] was assigning a particular color to a single number. This is very simple to formalize in Coq. We first show that we can fix the color of a particular number k – if a binary coloring of the natural numbers exists that does not make any Pythagorean triple monochromatic, then there must be one that assigns the chosen color to k: either the original one, or the one obtained by flipping the color assigned to every number. Using this result, we show that adding any (single) unit clause to the simplified formula obtained above does not break any of the properties proved earlier.

```
Lemma fix_one_color : ∀ (C:coloring), pythagorean_prop C →
  ∀ (n:positive) (b:bool), ∃ C', pythagorean_prop C' ∧ C' n = b.

Parameter TheBreak : positive.

Definition The_Asymmetric_CNF : CNF :=
  (pos TheBreak :: nil) :: simplified_BPT_formula TheN The_List.

Theorem symbreak_ok : unsat The_CNF ↔ unsat The_Asymmetric_CNF.

Theorem Pythagorean_Theorem'' :
  unsat The_Asymmetric_CNF → ∀ (C:coloring), ~pythagorean_prop C.

Theorem Pythagorean_dual_Theorem'' : ∀ C, satisfies C The_Asymmetric_CNF →
  ∃ C', pyth_lim_prop TheN C'.
```

To compute this formula, we again use program extraction to obtain a correct-by-construction OCaml program. As mentioned earlier, we obtain the list of numbers that guide the triple elimination process from the trace of the original simplification proof in [17], using an untrusted program. (Recall that the fact that this list is correct is immaterial to the soundness of the final result – although we want it to be correct in order to be able to reuse subsequent results from [17].) The value of `TheBreak` also needs to be instantiated to 2520.[5] The formula generated by our certified program and the CNF produced in [17] differ only in the identifiers assigned to each clause. Generating this formula takes approximately 35 minutes, with a peak memory consumption of 15.8 MB. The high computation time is due to the fact that the generation algorithm is optimized for the correctness proof, rather than for execution: these 35 minutes are dwarfed by the time required by later computation steps.

By instantiating the value of `TheN` to 7825 and simplifying the resulting satisfiable formula, we obtain a formula that is equivalent to the satisfiable CNF in [17]. To verify that this CNF is satisfiable using our development, we need two ingredients: a

[5] This value was originally chosen because 2520 is the number that occurs most often in the triples after simplification, hence the potential for simplification was highest.

procedure that, given a valuation and a CNF, decides whether the valuation satisfies the CNF, and a valuation that satisfies that CNF.

For the first point, we prove the following lemma in Coq, stating that it is decidable whether a valuation satisfies a CNF.

```
Lemma satisfies_dec : ∀ v c, {satisfies v c} + {~satisfies v c}.
```

Applying program extraction yields a program that implements the straightforward algorithm for checking satisfaction of a CNF: for each clause, check its literals in order until one is satisfied by the valuation, and then move to the next clause. If some clause does not contain any satisfied literal, immediately return `false`, otherwise return `true` when there are no more clauses.

For the second point, we read the valuation found in [17] using untrusted OCaml code and pass it to the OCaml function `satisfies_dec` extracted from the formalization. The call returns with `Left` after less than 1 second, indicating that the valuation indeed satisfies the simplified CNF for $n = 7825$. Soundness of program extraction and lemma `Pythagorean_dual_Theorem''` imply that there is a coloring of the natural numbers such that no Pythagorean triple containing only numbers less than or equal to 7824 is monochromatic.

4 Formalizing Cube-and-Conquer

To show that formula (1) derived above is unsatisfiable for $n = 7826$, the authors of [17] resorted to the cube-and-conquer methodology. In this section, we specify this methodology in Coq using our types of propositions, prove its soundness, and extract certified code that, given a propositional formula φ and a list of cubes, generates all the formulas that must be proved unsatisfiable to establish unsatisfiability of φ. This corresponds to the arrows to and from the boxed area in Figure 1.

We define a type `Cube` as a list of literals. Note that, while `Cube` and `Clause` are defined as the same type, literals in a cube are meant to be interpreted conjunctively. Thus, to add a cube to a CNF, we add all of its elements as individual clauses to the CNF.

```
Definition Cube := list Literal.

Fixpoint Cubed_CNF (F:CNF) (C:Cube) : CNF :=
  match C with
  | nil ⇒ F
  | l :: c ⇒ (l :: nil) :: (Cubed_CNF F c)
  end.
```

In order to apply the cube-and-conquer methodology, we need to have a set of cubes that is complete, i.e. any valuation satisfies at least one cube. This condition is equivalent to stating that the disjunction of the negations of all cubes is unsatisfiable. We construct this formula directly, by negating all the literals in each `Cube` and adding each resulting `Clause` to an empty `CNF`. The function `negate` implements negation on literals.

```
Definition negate (l:Literal) : Literal :=
  match l with
  | pos n ⇒ neg n
  | neg n ⇒ pos n
  end.
```

```
Definition neg_cube (C:Cube) : Clause := map negate C.

Definition noCube (C:list Cube) : CNF := map neg_cube C.
```

Soundness of cube-and-conquer is then very simple to state and prove: given a list of cubes and a formula `Formula`, if all the CNFs generated by the above functions are unsatisfiable, then so is `Formula`.

```
Lemma CubeAndConquer_lemma : ∀ (Formula:CNF) (Cubes:list Cube),
  (∀ (c:Cube), In c Cubes → unsat (Cubed_CNF Formula c)) →
  unsat (noCube Cubes) → unsat Formula.
```

Next, we can apply cube-and-conquer to `The_Asymmetric_CNF` and generate the formulas whose unsatisfiability we need to check. Again we extract the relevant functions (`Cubed_CNF` and `noCube`) to OCaml, relying on the soundness of program extraction.

We build the list of cubes in OCaml using untrusted code to process the files generated in [17]. The resulting list is passed as argument to `noCube`, and we check that the result coincides with the formula one used in the proof of unsatisfiability in [17] by using another function extracted from our development – namely, that from the lemma stating that equality of `CNF`s is decidable.[6] This step required 88.67s of CPU time and a peak memory consumption of 3.16 GB. Using exactly the same code to build the list of cubes, we can also reconstruct the one million formulas described in [17] independently (by iterating all cubes from the list and calling `Cubed_CNF`).

The only part in this process that is not formally verified is the chaining of arguments at the meta-level: we have reduced the problem of establishing unsatisfiability of `The_Asymmetric_Formula` to that of establishing unsatisfiability of 1,000,001 formulas, invoking lemma `CubeAndConquer_lemma`. However, the premise of `CubeAndConquer_lemma` quantifies over all cubes in the list. We need to trust that our implementation does test all these cubes, since we do not iterate over the list using extracted code. Although in principle this could be done formally, the time requirements make it unpractical: as we discuss in the next section, verifying all the unsatisfiability proofs is time-consuming and is only feasible if done in parallel. Still, each parallel computation starts by independently generating `The_Asymmetric_Formula` and then using it to generate a particular `Cubed_CNF`, which is then checked to be unsatisfiable.

5 Formalizing Unsatisfiability Proofs

The results in the previous section allow us to reduce the verification of non-existence of a solution to the Boolean Pythagorean Triples problem to that of checking unsatisfiability of 1,000,001 independent propositional formulas, originally established by an untrusted SAT solver that produced traces of its proofs. In this section, we show how we can rewrite these traces into lists of proof witnesses that allow us to reproduce the SAT solver's reasoning process in a certified checker formalized in Coq. This corresponds to the dotted arrows in Figure 1.

All the formal results on unsatisfiability so far were established using Coq. Differently, the results in this section rely also on the soundness of Coq's program extraction feature [24] (similarly to the verification of satisfiability of formula (1) with $n = 7825$ at the end of Section 3.4). Currently, this dependency seems impossible to overcome, as

[6] This is a sanity check, but it is nice that we can perform it using certified code.

the speed-up that is obtained by program extraction is essential to make these checks feasible. Hopefully, the CertiCoq project [1] will soon produce a certified compiler for Coq, which will increase the trustworthiness of program extraction.

5.1 The verification algorithm

All the unsatisfiability proofs we rely on can be verified using only reverse unit propagation. In order to check that this process is correct, we need to implement this process efficiently. Following ideas that are standard in theorem proving [3,11,15,23,30,31], we use an oracle that gives us additional information we can use to modify the algorithm behind reverse unit propagation, thereby improving its computational complexity. (Essentially, we provide references to the clauses that are used in this algorithm, in order to bypass the expensive process of finding them; we explain this in detail in § 5.3.)

The process of verifying that $F \models B$ by reverse unit propagation consists of iterating unit propagation on $F \wedge \neg B$ until the empty clause is derived. An equivalent dual formulation relies on keeping a set of literals D corresponding to the negated unit clauses derived in this process, initally starting with the literals in B. Removing these literals from a clause C corresponds to performing unit propagation using the clauses from D as unit clauses. If $C \setminus D$ is a unit clause, i.e., a single literal ℓ, we add $\bar{\ell}$ to D. If repeated application of set difference and addition of negated unit clauses to D results in $C \setminus D = \Box$ for some $C \in F$, we have shown that $F \models B$. Obviously, we reach $\Box$ using this approach if, and only if, $\Box \in F\downarrow_U$, due to confluence of $\rightarrow_U$.

This dual formulation allows for efficiently checking entailment of learned clauses when the sequence of clauses used is known. Assume that we are given B together with a set $\{C_1, \ldots, C_n\}\rangle$ of the clauses we should use at each step. Then we can compute the finite sequence $D_1, \ldots, D_n$ where $D_1 = \neg B$ and $D_{i+1} = D_i \cup (C_i \setminus D_i)$ for $1 < i < n$.

Lemma 4 (Soundness) *If $C_i \setminus D_i$ is a unit clause for all $1 \leq i < n$ and $D_n = \Box$, then $F \models B$.*

We omit the proof of this lemma, as it is formalized in Coq (lemma `propagate_true` on page 23).

Verifying unsatisfiability of a CNF now becomes an iterative process, where we start with an input CNF and iteratively add clauses justified by reverse unit propagation as suggested by an oracle. In order to be efficient, we keep a working set of clauses, which is initially empty; the oracle also allows us to add any clause from the input CNF to the working set and to delete clauses that are no longer needed.

5.2 Data structures for implementation

The algorithm above intensively repeats the same operations – computing set differences of two clauses and finding a clause in a CNF. (Observe that the clauses C_i in the premise of Lemma 4 must be in F: if they are provided by an untrusted oracle, then we need to be able to check that they indeed occur in F.) In order to implement it efficiently, we need to be able to execute these operations with as few resources as possible. To this end, we implement alternative representations of clauses and CNFs, which are not visible to the final user.

The first type we define is a new type TreeCNF, which implements CNFs as binary search trees of Clauses. We use this type to speed up the process of looking up clauses in the original CNF (which is extracted to a linked list) from linear to logarithmic complexity in the size of the CNF.

```
Definition TreeCNF := BinaryTree Clause.
```

The type TreeCNF uses the dependent type BinaryTree (described in [8]), which implements generic binary trees over any type that is equipped with a comparison operator. In particular, this type includes operations that work as expected if their arguments are binary search trees.

As we explain below, we use this datatype to store the input CNF, which we need to search intensively during verification. Transforming a CNF into a TreeCNF is done by function make_TCNF shown below: it iteratively adds each Clause to an initially empty TreeCNF using TCNF_add, which specializes the general function for adequately adding an element to a binary search tree by including information about how to compare two Clauses. Among other properties of this function, we show formally that it always returns a valid binary search tree. Soundness of program extraction guarantees that this property holds during execution of the extracted code.

```
Fixpoint make_TCNF (c:CNF) : TreeCNF :=
  match c with
  | nil ⇒ nought
  | cl :: c' ⇒ TCNF_add cl (make_TCNF c')
  end.

Lemma make_TCNF_wf : ∀ c, TCNF_wf (make_TCNF c).
```

Function make_TCNF does not attempt to balance the search tree constructed. In practice, the fact that the tree is not balanced is not a major issue, since the order of the clauses in the CNFs we consider yields trees with an acceptable depth.[7]

Finally, we define the reverse mapping from TreeCNF to CNF, which does a pre-order traversal of the tree and adds each element visited to an initially empty list of clauses. This function is declared as a coercion,[8] so that all notions previously defined for CNFs (e.g. unsatisfiability) automatically apply to TreeCNFs.

```
Fixpoint TCNF_to_CNF (c:TreeCNF) : CNF :=
  match c with
  | nought ⇒ nil
  | node cl c1 c2 ⇒ (cl :: TCNF_to_CNF c1) ++ TCNF_to_CNF c2
  end.

Coercion TCNF_to_CNF : TreeCNF ↣ CNF.
```

In order to compute set differences between clauses, we provide a similar alternative type for these objects.

```
Definition SetClause := BinaryTree Literal.
```

[7] Practical measurements show that these depths are at most approximately 150, for CNFs containing just under 15,000 clauses. Although this is much higher than for a perfectly balanced tree, it suffices for practical purposes, and allows us to avoid formalizing a balancing algorithm. See [8] for a similar discussion.

[8] Declaring a function of type A→ B as a coercion means that Coq applies it automatically whenever it expects a term of type B and one of type A is given. In order to guarantee termination, no composition of coercions may map a type to itself.

Again, we can build a `SetClause` from any `Clause` by adding its elements to an empty tree, one by one, using the appropriate operation on binary trees, so that the result is a binary search tree. The converse conversion, which again implements a pre-order traversal of the tree, is declared as a coercion, so that all notions we define for `Clause`s can again be transparently applied to `SetClause`s.

```
Fixpoint Clause_to_SetClause (cl:Clause) : SetClause :=
  match cl with
  | nil ⇒ nought
  | l :: cl' ⇒ SC_add l (Clause_to_SetClause cl')
  end.

Lemma C_to_SC_wf : ∀ (cl:Clause), SC_wf (Clause_to_SetClause cl).

Fixpoint SetClause_to_Clause (cl:SetClause) : Clause :=
  match cl with
  | nought ⇒ nil
  | node l cl' cl'' ⇒ l :: (SetClause_to_Clause cl') ++ (SetClause_to_Clause cl'')
  end.

Coercion SetClause_to_Clause : SetClause ↣ Clause.
```

Finally, we provide yet another implementation of CNFs, where we are allowed to address clauses by a pre-defined index.

```
Definition ICNF := Map {cl:SetClause | SC_wf cl}.
```

The type `ICNF` (for Indexed CNF) implements CNFs as Patricia trees of `SetClause`s that are binary search trees. Each element in the tree is paired with a natural number (the index) whose binary representation determines the path from the root to that element. The type `Map` is defined in Coq's standard library, while condition `SC_wf cl` abbreviates the property that `cl` respects the well-formedness condition of binary search trees (see [8]). Note that these proofs of well-formedness carry no computational meaning, and are ignored by program extraction (see Section 5).

The type `ICNF` is used to store the working set used during verification. Unlike the type `BinaryTree`, the type `Map` supports efficient deletion, which is instrumental to keep the working set small. Furthermore, accessing a clause by index is faster, since we avoid comparisons with the clauses at each node on the path we follow.

We can convert any `ICNF` to a `CNF` by ignoring the indices assigned to the clauses, converting each `SetClause` to a `Clause`, and adding them one by one to an empty tree. The result is a well-formed `TreeCNF`. We omit the definition of this function `ICNF_to_CNF`, as it requires understanding the underlying type `Map`; it is again defined as a coercion.

5.3 The formalized checker

We recall the possible actions we need in order to verify a proof of unsatisfiability: delete a clause from the working set; add a clause from the input CNF to the working set; and add a clause to the working set that is entailed by it, and such that this entailment can be verified by reverse unit propagation. We formalize these actions as a type `Action` with three constructors, corresponding to the three possible operations.

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

Since we are using OCaml as the extraction language and the proofs we want to check are typically too large to fit in memory, we need to make the oracle lazy. We do this by defining a Coq type of lazy lists, defined in the same way as lists (with constructors `lnil` and `lcons`), but where the second argument of `lcons` is guarded by an identity function. On extraction, these functions are mapped to the adequate OCaml constructs implementing laziness.[9]

```
Definition LazyT := id (A:=Type).

Inductive lazy_list (A:Type) :=
    lnil : lazy_list A
  | lcons : A → LazyT (lazy_list A) → lazy_list A.

Definition Oracle := LazyT (lazy_list Action).
```

The core of our development is the propagation step, which is implemented as a function `propagate`. We first describe this function informally. This function takes three arguments: the working set $\overrightarrow{c}$, a clause c (initially, the clause that we want to add), and a list $\overrightarrow{i}$ of indices of clauses. We iteratively go through the list of indices, returning `false` if this list finishes before we obtained the empty clause. We extract the clause with the current index (i_0) from $\overrightarrow{c}$, which we denote $\overrightarrow{c}(i_0)$, and compute the set difference between it and the argument clause. If this results in the empty clause, we return `true`; otherwise, we check that there is only one element ℓ in this set (returning `false` otherwise) and recur after extending the current clause with $\bar{\ell}$.

Algorithm 1 Function PROPAGATE

```
1: function PROPAGATE($\overrightarrow{c}, c, \overrightarrow{i}$)
2:     if $\overrightarrow{i} = \emptyset$ then return FALSE
3:     else if $\overrightarrow{c}(i_0) \setminus c = \emptyset$ then return TRUE
4:     else if $\overrightarrow{c}(i_0) \setminus c = \{\ell\}$ then return PROPAGATE($\overrightarrow{c}, \{\bar{\ell}\} \cup c, \overrightarrow{i}$)
5:     else return FALSE
6:     end if
7: end function
```

For completeness we include the corresponding Coq definition, implemented as function `propagate`. Clause `c` is represented as a set (i.e. with type `SetClause`). Extracting the clause with the current index from `cs` is done by function `get_ICNF`, and computing set differences by `SC_diff`. All tests are done by elimination over applications of lemma `SetClause_eq_nil_cons`, which states that we can decide whether a binary tree is empty or not, and in the latter case returns the element in the root node and its two descendants: the first test checks whether the set difference computed is the empty clause; in the negative case we obtain the element in the root (`l`) and the two subtrees (`c'` and `c''`; there is a spurious proof term that is ignored). If either `c'` or `c''` differs from the empty tree we return `false` (line 5 of the above algorithm), otherwise the function recurs as in line 4 of the above algorithm.

```
Fixpoint propagate (cs: ICNF) (c: SetClause) (is:list ad) : bool :=
```

[9] In principle, this approach could break soundness of extraction; we are using the fact that these constructs behave as identities. Targeting a lazy language like Haskell would not require this workaround; however, our experiments showed that using OCaml instead of Haskell reduced computation times to around one-fourth.

```
match is with
| nil ⇒ false
| (i:: is) ⇒ match SetClause_eq_node_nought (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_node_nought c' with
                | inleft _ ⇒ false
                | inright _ ⇒ let (c'',_) := Hc in
                              match SetClause_eq_node_nought c'' with
                              | inleft _ ⇒ false
                              | inright _ ⇒ propagate cs (SC_add (negate l) c) is
end end end end.
```

For convenience, soundness of this function is split into several lemmas. If the inner set difference returns □, then the argument clause is entailed by the working set; if it returns a singleton, then we can show that the working set entails the argument clause by checking that it entails the extended clause we construct. Iterating the latter result allows us to conclude that, if (`propagate cs c is`) returns `true`, then `cs` entails `c`. These lemmas are all stated in terms of `TreeCNF`s, as this is the format in which we will store the input CNF. (Technically, we add the true clause $x_1 \vee \overline{x_1}$ to `cs`: this is necessary because this clause is returned if we search for an index that does not correspond to a clause in the working set. This default case is needed because Coq does not allow functions to be partially defined.)

```
Lemma propagate_empty : ∀ (cs:TreeCNF) (c c':SetClause),
  SC_diff c c' = nought → entails (TCNF_add c cs) c'.

Lemma propagate_singleton : ∀ (cs:TreeCNF) (c c':SetClause), ∀ (l:Literal),
  entails cs (SetClause_to_Clause (SC_add (negate l) c')) →
  SC_diff c c' = (node l nought nought) → entails (TCNF_add c cs) c'.

Lemma propagate_true : ∀ (cs:TreeCNF) (c:SetClause),
  entails (TreeCNF_add true_SC cs) c → entails cs c.
```

Lemma `propagate_true` formalizes the informal Lemma 4 above.

Using `propagate`, we now define a function `refute` that processes an arbitrary oracle. Again we begin by presenting the underlying algorithm informally. Recall that an oracle is a list of actions; we denote by *hd* and *tl* the usual head and tail functions on lists.

Starting with the empty working set (a set of clauses accessible by index), this function sequentially applies each action indicated by the oracle; if the oracle ends, the result is `false` (the empty clause was not derived). In the case of a deletion, all clauses with indices in the given set are removed from the working set. Action `O` is processed by first checking that the clause given occurs in the original CNF; if this is not the case, the refutation fails. New clauses are added by checking that they follow by reverse unit propagation using procedure `propagate`; if the new clause is the empty clause, then `refute` returns `true`, otherwise the new clause is added and the algorithm recurs.

The Coq definition of `refute` follows this structure closely, but is complicated by the need for some explicit type convertions and to ensure laziness. Function `refute` receives an argument of type `CNF`, converts it to a `TreeCNF`, initializes the working set, and calls the auxiliary function `refute_work`. This function in turn does case analysis on each action to perform the corresponding operation on the working set. The function `force`, defined as the identity, is used to force evaluation of its argument in the extracted lazy implementation. The proof term `make_TCNF_wf c`, stating that the `TreeCNF` constructed initially is a well-formed binary search tree, is necessary to prove its soundness, but it

Algorithm 2 Function `refute`

```
function REFUTE($c, \overrightarrow{o}$)
    return REFUTE_WORK($\emptyset, c, \overrightarrow{o}$)
end function

function REFUTE_WORK($w, c, \overrightarrow{o}$)
    if $\overrightarrow{o} = \emptyset$ then return FALSE
    else if $hd(\overrightarrow{o}) = (D\ \overrightarrow{i})$ then
        return REFUTE_WORK($w \setminus \{\varphi_i \mid i \in \overrightarrow{i}\}, c, tl(\overrightarrow{o})$)
    else if $hd(\overrightarrow{o}) = (O\ i\ cl)$ then
        if $cl \in c$ then return REFUTE_WORK($w \cup \{i \mapsto cl\}, c, tl(\overrightarrow{o})$)
        else return FALSE
        end if
    else if $hd(\overrightarrow{o}) = (R\ i\ \emptyset\ \overrightarrow{i})$ then return PROPAGATE($w, \emptyset, \overrightarrow{i}$)
    else if $hd(\overrightarrow{o}) = (R\ i\ cl\ \overrightarrow{i})$ then
        return PROPAGATE($w, cl, \overrightarrow{i}$)$\wedge$REFUTE_WORK($w \cup \{i \mapsto cl\}, c, tl(\overrightarrow{o})$)
    end if
end function
```

is removed by program extraction and therefore never constructed. For readibility, we abbreviate some other unextracted proof terms in the code below to an underscore.

```
Definition refute (c:CNF) (o:Oracle) : Answer :=
  refute_work empty_ICNF (make_TCNF c) (make_TCNF_wf c) o.

Function refute_work (w:ICNF) (c:TreeCNF) (Hc:TCNF_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.
```

Soundness of `refute_work` and `refute` is established in the following result: 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 either the incorrect data can be ignored, or `refute` outputs `false`.)

```
Theorem refute_correct : ∀ (c:CNF) (o:Oracle), refute c o = true → unsat c.
```

5.4 Verifying the Boolean Pythagorean Triples conjecture

To finish our proof, we need to verify that the 1,000,001 formulas generated at the end of the previous section (the one million cubes and the negation of their disjunction) are unsatisfiable. This requires preparing 1,000,001 oracles in the appropriate format, which are then given as argument to the code extracted from the formalization (function `refute` above) by means of a handwritten interface. In order to produce the oracles, we reconstituted the original unsatisfiability proofs from [17] using the incremental SAT solver iGlucose within 508 CPU days and ran it through the drat-trim version

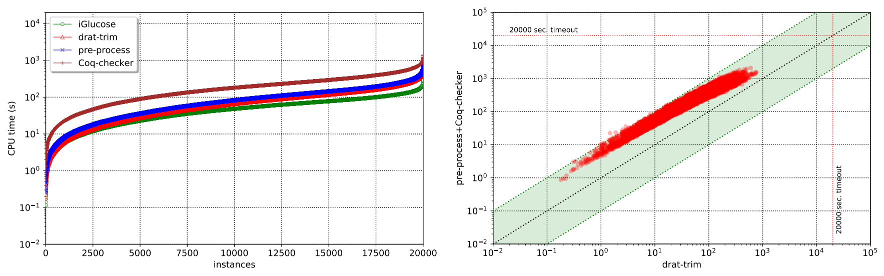

Fig. 2 **Left:** Cactus plot comparing the runtime of the incremental SAT solver (iGlucose), an uncertified checker (drat-trim), the pre-processing of the proofs with the drat-trim version from [9] (pre-process), and the certified extracted checker (Coq-checker) on 20,000 cubes (2%). **Right:** Scatter plot comparing the runtime of an uncertified checker (drat-trim) and of the certified extracted checker including pre-processing (pre-process+Coq-checker).

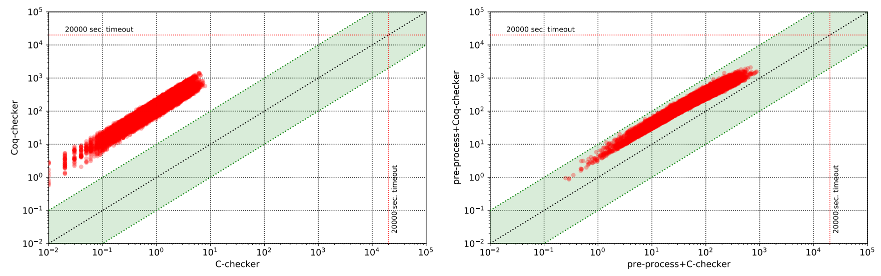

Fig. 3 Scatter plots comparing the runtime of an uncertified C implementation using the same oracle information (C-checker) and of the extracted certified checker (Coq-checker) without (**left**) and with (**right**) pre-processing on 20,000 cubes (2%).

from [9], which required an additional 871 CPU days. The result of this process is an oracle in the format described earlier. Thus, the total untrusted generation of proofs took 1,379 CPU days. Verifying the 1,000,001 individual proofs with the certified extracted checker took 2,608 CPU days, i.e., approximately twice as long.

To make an empirical evaluation of the performance of the extracted certified checker and compare it to uncertified alternatives, we instrumented the extracted code to capture detailed timing data for the individual cubes and reran on 20,000 cubes, i.e., on 2% of the total cubes. Figure 2 includes on its left a cactus plots showing that the runtimes of the SAT solver, of drat-trim as an uncertified checker, of the pre-processing with the drat-trim version from [9], and the extracted certified checker lie within one order of magnitude from each.[10] On the right, a scatter plot compares the runtime of drat-trim as an uncertified checker with the tool chain consisting of the modified drat-trim and the extracted certified checker, showing that the runtimes for virtually all cubes are within one order of magnitude, and that for the computationally more demanding cubes the overhead is only around 100%.

To assess the overhead introduced by certification, we ran an uncertified checker implemented in C on the same oracle data generated by the modified drat-trim version

[10] The modified version of drat-trim from [9] is slightly slower than the original, due to the need to generate more detailed proofs.

for the same 20,000 cubes. Figure 3 compares the runtimes of these two implementations of the same algorithm, showing on the left that the uncertified C checker is approximately two orders of magnitude faster. This comparison is meaningless in the big picture, though, as neither can be run without the pre-processing step generating the oracle data. When including that step, the scatter plot on the right shows that the overall overhead is strictly less than one order of magnitude. Note that the overhead introduced appears to be independent of how computationally demanding the cubes are. We verified that the reason for this is an (empirically) linear relationship between the sizes of the oracles and the runtimes of both the uncertified and the extracted certified checker; this relationship can also be deduced by analysing Algorithm 5.3.

The extracted checker is nearly pure. Aside from lazy lists, which were discussed earlier, we also extracted 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 customary optimization 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.

As such, we claim that our formalization provides a more trustworthy proof of the Boolean Pythagorean Triples problem. Besides a formal proof of the soundness of the encoding and of the cube-and-conquer methodology, all propositional formulas are generated by trusted code and directly shown unsatisfiable by a certified checker, without the need to store them in the file system and combine them through *ad-hoc* scripts. Our dependence on the soundness of Coq is limited, since we only use a restricted subset of the language that has widely been considered correct and stable for over a decade. By contrast, the authors of [17] rely on the soundness of drat-trim for checking unsatisfiability of propositional formulas. Although drat-trim consists of only around 1,000 lines of C-code, its correctness is by no means given: both [7] and [21] report instances of wrong proofs of unsatisfiability that were accepted by drat-trim, while [27] and [28] raise more general concerns regarding its semantics.

Chronologically, this work was done in the reverse order to the presentation. This means that the verification of the unsatisfiability proofs described in [17] was done starting from the CNFs provided by those authors, rather than from the ones generated by our encoding. As such, we performed an additional verification step, checking that the CNFs we generate are equivalent (as sets of sets) to those used in the validation of the unsatisfiability proofs. This step was also done by correct-by-construction extracted code, whose soundness was proved in Coq; we omit its description, as any independent verification of our results can follow the order we described.

The Coq development consists of 85 definitions and 121 lemmas/theorems, amounting to 1,946 lines with a total file size of 55.6 kB. The relative sizes of each part of the development is given in Table 1. This development reuses the implementation of Patricia trees from the Coq standard library, as well as the implementation of binary trees from the development described in [8]. Most proofs follow the mathematical proofs or informal arguments described in previous work [5, 17, 18], and did not pose significant challenges. Soundness of the new algorithm for efficiently verifying reverse unit propagation, described in Section 5, was also simple to prove in Coq. This is not surprising, since most results describe properties of propositional logic; the complexity of SAT solving derives mostly from the complexity of the algorithms required to solve the SAT problem efficiently, which is an aspect orthogonal to our development.

The whole formalization is available at `http://imada.sdu.dk/~petersk/bpt/`.

topic	definitions	lemmas	lines	size (kB)
propositional logic and CNFs (§ 3.1)	21	29	340	9.89
formalizing the BPT problem (§ 3.2–3.4)	26	42	648	18.49
cube-and-conquer (§ 4)	4	4	73	1.75
unsatisfiability checker (§ 5)	32	42	855	25.56

Table 1 Size of the Coq formalization

6 Conclusions

We described a formal verification of the proof of inexistence of a binary coloring of the natural numbers such that no Pythagorean triple is monochromatic, thus confirming partition regularity for $k = 2$ of the Pythagorean equation, established by Heule *et al.* in 2016 [17].

Our formalization consists of: stating the problem in Coq; defining the propositional encoding from [5] and proving that it is correct; reproducing the mathematical argument behind the simplification of the formula thus obtained and proving its soundness; formalizing the methodology of cube-and-conquer in Coq and proving its correctness; and formalizing the process of verifying unsatisfiability proofs for propositional formulas based on reverse unit propagation.

From our development we extract correct-by-construction OCaml code that takes input from the development in [17] and verifies that the proof constructed by those authors indeed establishes the inexistence of a solution to the Boolean Pythagorean Triples problem. While theoretically the whole verification could be done inside Coq, the time and memory requirements for such a task make it unfeasible. Using program extraction allows us to complete the task in around 13 CPU years, with very little compromise on the level of confidence guaranteed by the theorem prover.

While we focused exclusively on the Boolean Pythagorean Triples problem, our formalization includes two general-purpose components that can be used independently: the framework for dividing a propositional formula into cubes (given the list of cubes generated independently); and the unsatisfiability checker, which can be used to verify unsatisfiability of other propositional formulas. As a consequence, we provide a powerful framework in which we can, in principle, rigorously verify proofs of other open conjectures in Mathematics that are obtained via an encoding into SAT. Future such applications will only need to formalize the problem and its propositional encoding, and can afterwards apply our tools to verify results produced by SAT solvers running on the generated formulas.

Acknowledgements The authors would like to thank Marijn Heule for providing some valuable insights, namely regarding the mathematical interpretation of the simplification procedure used in [17]. This work was partially supported by the Danish Council for Independent Research, Natural Sciences, grant DFF-1323-00247.

References

1. Anand, A., Appel, A.W., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Bélanger, O.S., Sozeau, M., Weaver, M.: CertiCoq: A verified compiler for Coq. In: CoqPL Workshop (2017)

2. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Texts in Theoretical Computer Science. Springer (2004)
3. Blanqui, F., Koprowski, A.: CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. Math. Struct. Comp. Sci. **21**, 827–859 (2011)
4. Cook, S.A.: The complexity of theorem-proving procedures. In: M.A. Harrison, R.B. Banerji, J.D. Ullman (eds.) STOC, pp. 151–158. ACM (1971)
5. Cooper, J., Overstreet, R.: Coloring so that no pythagorean triple is monochromatic. CoRR **abs/1505.02222** (2015)
6. Coquand, T., Huet, G.P.: The calculus of constructions. Inf. Comput. **76**(2/3), 95–120 (1988)
7. Cruz-Filipe, L., Heule, M.J.H., Jr., W.A.H., Kaufmann, M., Schneider-Kamp, P.: Efficient certified RAT verification. In: de Moura [26], pp. 220–236
8. Cruz-Filipe, L., Larsen, K.S., Schneider-Kamp, P.: Formally proving size optimality of sorting networks. J. Autom. Reasoning (accepted for publication)
9. Cruz-Filipe, L., Marques-Silva, J., Schneider-Kamp, P.: Efficient certified resolution proof checking. In: TACAS, *LNCS*, vol. 10205. Springer (2017)
10. Cruz-Filipe, L., Schneider-Kamp, P.: Formally verifying the boolean pythagorean triples conjecture. In: Eiter and Sands [14], pp. 509–522
11. Cruz-Filipe, L., Wiedijk, F.: Hierarchical reflection. In: K. Slind, A. Bunker, G. Gopalakrishnan (eds.) TPHOLs, *LNCS*, vol. 3223, pp. 66–81. Springer (2004)
12. Darbari, A., Fischer, B., Marques-Silva, J.: Industrial-strength certified SAT solving through verified SAT proof checking. In: ICTAC, *LNCS*, vol. 6255, pp. 260–274. Springer (2010)
13. Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem-proving. Communications of the ACM **5**, 394–397 (1962)
14. Eiter, T., Sands, D. (eds.): LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, 7-12th May 2017, *EPiC Series*, vol. 46. EasyChair (2017)
15. Fouilhé, A., Monniaux, D., Périn, M.: Efficient generation of correctness certificates for the abstract domain of polyhedra. In: F. Logozzo, M. Fähndrich (eds.) SAS, *LNCS*, vol. 7935, pp. 345–365. Springer (2013)
16. Goldberg, E.I., Novikov, Y.: Verification of proofs of unsatisfiability for CNF formulas. In: DATE, pp. 10,886–10,891 (2003)
17. Heule, M., Kullmann, O., Marek, V.W.: Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In: N. Creignou, D. Le Berre (eds.) SAT, *LNCS*, vol. 9710, pp. 228–245. Springer (2016)
18. Heule, M., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In: K. Eder, J. Lourenço, O. Shehory (eds.) HVC, *LNCS*, vol. 7261, pp. 50–65. Springer (2012)
19. Järvisalo, M., Biere, A., Heule, M.: Blocked clause elimination. In: J. Esparza, R. Majumdar (eds.) TACAS, *LNCS*, vol. 6015, pp. 129–144. Springer (2010)
20. Konev, B., Lisitsa, A.: Computer-aided proof of erdős discrepancy properties. Artif. Intell. **224**, 103–118 (2015)
21. Lammich, P.: Efficient verified (UN)SAT certificate checking. In: de Moura [26], pp. 237–254
22. Landman, B.M., Robertson, A.: Ramsey Theory on the Integers, *The Student Mathematical Library*, vol. 24. AMS (2004)
23. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009)
24. Letouzey, P.: Extraction in Coq: An overview. In: A. Beckmann, C. Dimitracopoulos, B. Löwe (eds.) CiE 2008, *LNCS*, vol. 5028, pp. 359–369. Springer (2008)
25. Mijnders, S., de Wilde, B., Heule, M.: Symbiosis of search and heuristics for random 3-SAT. In: D.G. Mitchell, E. Ternovska (eds.) LaSh (2010)
26. de Moura, L. (ed.): Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings, *LNCS*, vol. 10395. Springer (2017)
27. Philipp, T., Rebola-Pardo, A.: Towards a semantics of unsatisfiability proofs with inprocessing. In: Eiter and Sands [14], pp. 65–84
28. Rebola-Pardo, A., Biere, A.: Two flavours of DRAT. In: Pragmatics of SAT 2018 (accepted for publication)

29. Silva, J.P.M., Sakallah, K.A.: Conflict analysis in search algorithms for satisfiability. In: ICTAI, pp. 467–469. IEEE Computer Society (1996)
30. Sternagel, C., Thiemann, R.: The certification problem format. In: C. Benzmüller, B.W. Paleo (eds.) UITP, *EPTCS*, vol. 167, pp. 61–72 (2014)
31. Wetzler, N.D., Heule, M.J., Hunt Jr., W.A.: Mechanical verification of SAT refutations with extended resolution. In: S. Blazy, C. Paulin-Mohring, D. Pichardie (eds.) ITP, *LNCS*, vol. 7998, pp. 229–244. Springer (2013)
32. Wiedijk, F. (ed.): The Seventeen Provers of the World, *LNCS*, vol. 3600. Springer (2006)