Formally Proving the Boolean Pythagorean Triples Conjecture

Luís Cruz-Filipe and Peter Schneider-Kamp

Department of Mathematics and Computer Science,
University of Southern Denmark, Odense, Denmark
{lcf,petersk}@imada.sdu.dk

Abstract

In 2016, Heule, Kullmann and Marek solved the Boolean Pythagorean Triples problem: is there a binary coloring of the natural numbers such that every Pythagorean triple contains an element of each color? By encoding a finite portion of this problem as a propositional formula and showing its unsatisfiability, they established that such a coloring does not exist. Subsequently, this answer was verified by a correct-by-construction checker extracted from a Coq formalization, which was able to reproduce the original proof. However, none of these works address the question of formally addressing the relationship between the propositional formula that was constructed and the mathematical problem being considered. In this work, we formalize the Boolean Pythagorean Triples problem in Coq. We recursively define a family of propositional formulas, parameterized on a natural number n, and show that unsatisfiability of this formula for any particular n implies that there does not exist a solution to the problem. We then formalize the mathematical argument behind the simplification step in the original proof of unsatisfiability and the logical argument underlying cube-and-conquer, obtaining a verified proof of Heule *et al.*'s solution.

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 an instance of an important family of problems in Ramsey theory on the integers [16]: 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*. Schur's theorem, van der Waerden's theorem and Rado's theorem all establish partition regularity of particular equations.

Regularity of the Pythagorean equation was finally established in 2016, when Heule, Kullmann and Marek [13] showed that it is already impossible to partition the set $\{1, \ldots, 7825\}$ into two sets such that none of them contains 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 [6]), which was then simplified and solved using the cube-and-conquer method [14].

More precisely, the propositional formula considered is

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

where each x_i is a propositional variable and $\bar{\cdot}$ denotes logical negation. This formula exhaustively lists the 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 coloring of the natural numbers without monochromatic Pythagorean triples.

This formula was first simplified using blocked clause elimination and symmetry breaking – removing and adding clauses in a way that preserves (un)satisfiability.[1] In this particular case, blocked clause elimination can be explained in a mathematically meaningful way: it corresponds to removing from a list of triples L every triple (a, b, c) containing a number that does not occur in any other triple. Indeed, if e.g. a does not occur in any other triple in L, and we can partition the natural numbers such that none of the remaining triples in L is monochromatic, then we can extend such a coloring to L by eventually changing the color of a. By iterating this argument over the set of Pythagorean triples in $\{1, \ldots, 7825\}^3$, 2136 of the original 9472 triples can be ignored. The symmetry break applied at the end consists of assigning a particular color to a single number (in this case, 2520 was assigned to true).

The second step was dividing the problem into one million *cubes*: a set of partial assignments that cover the whole space of possible valuations on the 3745 propositional variables actually used. 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.

However, trusting that a formula is unsatisfiable simply because of the result of a SAT solver is not completely satisfactory, as there is no guarantee that the SAT solver is correct. For this reason, the authors also produced a trace of their unsatisfiability proofs that they verified using an independently written solver, 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 next step was therefore to formalize the process of checking unsatisfiability proofs using the trace provided by the SAT solver within the theorem prover Coq [9]. Using Coq's extraction mechanism (an implementation of the Curry–Howard isomorphism between the Calculus of Construction, Coq's underlying type theory, and the programming language OCaml), this yielded a similar checker whose correctness is now guaranteed by the soundness of the theorem prover and of the extraction mechanism. In the first stage, this checker was only able to verify proofs using a sublanguage of the trace format, and as a consequence was only able to validate the claims of unsatisfiability (see Figure 1). The checker was subsequently extended to the full trace format [8], allowing also the simplification step to be checked.

Taken together, these independent verifications constitute a formal proof that the propositional formula (1) is unsatisfiable. However, the mathematical argument connecting this result to the original formulation of the Boolean Pythagorean Triples has not been formalized. To understand why this can be seen as a problem, we point out the places in the process where we still rely on soundness of informal arguments.

1. The Boolean Pythagorean Triples problem regards colorings of the natural numbers, whereas formula (1) only addresses a finite subset of these.
2. Formula (1) was generated using a C program whose soundness was never discussed.
3. The metalevel argument for soundness of cube-and-conquer requires manipulating formulas too large to process by hand, and have to be combined using e.g. command-line tools.

[1] To show partition-regularity of the Pythagorean equation, only preservation of satisfiability is required; however, preservation of unsatisfiability is also discussed in [13], and used to show that 7825 is the smallest n for which the set $\{1, \ldots, n\}$ cannot be 2-colored without including a monochromatic Pythagorean triple.

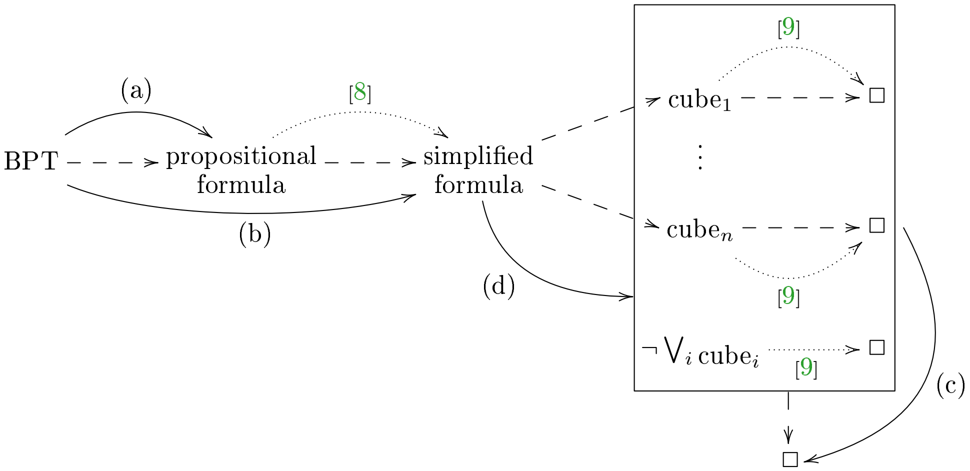

Figure 1: The original proof and the different verification steps. The dashed arrows denote the steps in the original proof [13]: a first propositional formula was generated by a C program, and subsequently simplified, divided and solved by SAT solvers. The dotted arrows denote proofs of unsatisfiability obtained by a SAT solver that were verified by a certified checker extracted from a Coq formalization [8, 9]. The solid arrows denote the contribution of this work: the generation, in Coq, of propositional formulas that are proved to represent the original mathematical problem, directly (a) and after simplification (b); the formal specification of the simple reasoning behind cube-and-conquer (c); and the generation of the formulas that are given as input to cube-and-conquer (d).

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 C program that generates the formula is also very easy to check for correctness; 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. In particular, we use the theorem prover Coq to:

- state the Boolean Pythagorean Triples problem as a logic formula;
- generate a family of propositional formulas (parameterized on n) and show that unsatisfiability of any formula in this family implies that the Boolean Pythagorean Triples problem does not have a solution;
- formalize soundness of cube-and-conquer;
- formalize the generation of all the formulas whose unsatisfiability is required by cube-and-conquer (parameterized on the individual cubes).

We also formalize the mathematical argument behind the simplification of formula (1), so that we directly generate the simplified formula (see Figure 1).

Due to the huge size of the formulas and trace files involved, it is impossible to perform the whole verification process inside Coq; thus, we use program extraction to obtain code that is correct by construction, and we rely on metalevel reasoning to chain the different steps in the process. However, we reduce this dependency to checking that the same arguments are provided to different functions. To avoid repeating the expensive verification steps in [9], we use the entailment checking capability of the checker from [8] to reuse those results.

This paper is organized as follows. We briefly summarize relevant concepts in Section 2. In Section 3 we describe the formalization, in Coq, of the Boolean Pythagorean Triples problem and of its relationship to formula (1). Section 4 describes the formalization of the simplification step in Coq. The soundness of cube-and-conquer and its application to this problem is formalized in Section 5. We conclude in Section 6.

The full development is available online [10].

2 Background and Related Work

We work with propositional logic over a countable set of propositional symbols $\{x_i\}_{i\in\mathbb{N}}$. A *literal* is a propositional symbol x_i or its negation, which we denote by $\overline{x_i}$. A clause is a disjunction of literals, and a CNF (conjunctive normal form) is a conjunction of clauses. It is well known that every propositional formula is equivalent to a CNF, and we therefore restrict ourselves to this fragment of the language. We denote the empty clause by $\Box$, and observe that every clause is a CNF. We use ℓ to range over literals, c to range over clauses, and C to range over CNFs.

A *valuation* is a function $V : \{x_i\}_{i\in\mathbb{N}} \to \mathbb{B}$, where $\mathbb{B} = \{\top, \bot\}$ is the set of Boolean truth values. Often we identify propositional symbols with the natural numbers via the trivial isomorphism $i \leftrightarrow x_i$, and view valuations as functions from $\mathbb{N}$ to $\mathbb{B}$. We say that V *satisfies* a propositional symbol x_i, denoted $V \models x_i$, if $V(x_i) = \top$, and that V satisfies a negative literal $\overline{x_i}$ if $V(x_i) = \bot$. Satisfaction extends to CNFs in the usual way: $V \models \bigvee_{j\in J} \ell_j$ if $V \models \ell_j$ for some $j \in J$, where each ℓ_j is a literal, and $V \models \bigwedge_{j\in J} c_j$ if $V \models c_j$ for all $j \in J$, where each c_j is a clause. A CNF C is *satisfiable* if there exists a valuation V such that $V \models C$; otherwise C is *unsatisfiable*. In particular, the empty clause $\Box$ is unsatisfiable. The *Boolean satisfiability problem* (SAT) consists of determining whether a given CNF is satisfiable; this problem is known to be NP-complete. Nevertheless, state-of-the-art SAT solvers are able to solve extremely large instances of this problem, and there have been a number of recent successes in encoding mathematical problems as propositional formulas and solving these automatically [15, 4, 5].

There are two problems with this approach. The first is ensuring soundness of the encoding: how can one guarantee that (un)satisfiability of a particular propositional formula implies a particular mathematical statement? This aspect is usually not given much emphasis, because propositional encodings tend to be very straightforward and “obviously” correct.

A more serious drawback of SAT solvers regards the reliability of their answers. These programs are usually extremely complex and impossible to prove correct by mathematical methods. When a formula is claimed to be satisfiable, the SAT solver usually returns a satisfying assignment that can be verified independently; however, in the case of unsatisfiability, no such witness can be produced. Recently, an effort has been made to have SAT solvers return traces of unsatisfiability proofs – enough information that is, in principle, sufficient to reproduce the proof independently by another, simpler program (which still needs to be trusted or proved correct by formal methods) [3, 18, 1, 13]. A more ambitious effort was the development of a more expressive format that enables formalization of the whole proof-checking process within a formal theorem prover [9]. This format, called GRIT (Generalized ResolutIon Trace format), is able to represent resolution proofs based on reverse unit propagation. This checker was later extended to the more expressive LRAT format [8], which also allows for steps based on the RAT property [12]. For the purpose of this work, both the details of these formats and the logical properties they encode are immaterial.

Both [8, 9] and the current development we use the 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, once the proofs have been built, they can be checked automatically: type checking in these languages is decidable, and implementable by a small kernel that is simple enough to be checked for correctness. The remaining interface does not need to be trusted: if an error in the program allows a wrong proof-term to be build, 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. [19] for an overview of the major theorem provers currently in use.)

The type theory underlying Coq is the Calculus of Constructions [7], a dependent type theory with inductive and co-inductive types. We highlight some of the most relevant features of the Calculus of Constructions. The logic corresponding to this theory 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). This allows an implementation of a realizability interpretation in terms of program extraction [17], 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.

The formalization of the process of verifying unsatisfiability proofs in [9] uses a deep embedding of CNFs into Coq's type theory. We summarize the key aspects that are relevant for this presentation. Literals are defined as elements of an inductive type `Literal` with two constructors `pos`,`neg`:`positive` → `Literal`, where `positive` is the type of positive integers (with a binary representation, designed to be efficient under program extraction). The intended meaning is that (`pos n`) corresponds to the positive literal x_n, and (`neg n`) to the negative literal $\overline{x_n}$. Likewise, the type `Valuation` of valuations is the function type `positive` → `bool` (where `bool` is the Coq type of Booleans) with the natural semantics. Clauses (type `Clause`) are lists of literals, and CNFs are binary trees of Clauses. We use two different implementations of binary trees: a standard implementation of binary search trees developed originally as part of the proof in [11], using lexicographic ordering on clauses as the underlying order, that yields a type `CNF`, which we use to state most results; and an implementation from Coq's standard library where elements are referenced by means of a key that is a positive integer whose binary representation denotes the path to the element, which yields an efficient datatype `ICNF` used in the extracted program.

Many definitions in our formalization are made by case analysis on whether a particular property holds. In Coq, properties are typically formalized using the type `Prop`, over which terms with computational content cannot depend. To bypass this limitation, there are special types for expressing *decidability* results: lemmas stating that one can decide whether a given property holds or not. A typical example is the (dependent) type `sumbool`; in Coq notation, the formula `{A}+{B}` (where `A` and `B` are themselves of type `Prop`) has type (`sumbool A B`), and terms inhabiting this type can be of the form (`left pA`) with `pA:A` or (`right pB`) with `pB:B`. Program extraction forgets the proof terms `pA` and `pB`, but not the constructor. This allows us to write definitions by case analysis over such terms using an if-then-else syntax.

All experiments were 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.

3 Encoding the Boolean Pythagorean Triples Problem in Coq

Our formalization is built upon the Coq type `positive` of binary natural numbers. Colorings are functions from this type to the type of booleans: letting the two colors be `true` and `false` simplifies the development, since this is also type of valuations. A Pythagorean triple is defined as expected. A coloring has the positive Pythagorean property (`pythagorean_pos`) if every pythagorean triple contains two elements of different colors. Equivalently, it has the negative Pythagorean property (`pythagorean_neg`) if any monochromatic triple is not Pythagorean. Since equality on the natural numbers is decidable, these properties are intuitionistically equivalent.

```
Definition coloring := positive → bool.

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

Definition pythagorean_pos (C:coloring) :=
  ∀ a b c, pythagorean a b c → (C a <> C b) ∨ (C a <> C c) ∨ (C b <> C c).

Definition pythagorean_neg (C:coloring) :=
  ∀ a b c, C a = C b → C a = C c → ~pythagorean a b c.

Theorem pythagorean_equiv : ∀ C, pythagorean_pos C ↔ pythagorean_neg C.
```

We now construct a family of propositional formulas, parameterized on n, corresponding to formula 1 with a variable upper bound. We first construct a list of all Pythagorean triples with elements in $\{1,\ldots,n\}^3$ by double iteration. To simplify the recursion, the recursive argument is a unary natural number (type `nat`).

In `inner_cycle` below, we first compute `sqrt` to be the integer square root of `n*n+mN*mN` (`Pos.of_nat` is the mapping between the two different representations of natural numbers, so `mN` is `m` as a binary integer), and then check whether these three values are in a Pythagorean triple (using the fact that this is a decidable property, as expressed by `pythagorean_dec`). In the affirmative case, we add the triple (`mN`,`n`,`sqrt`) to the result; then we recur. At the end, (`inner_cycle n m b`) returns a list of all Pythagorean triples whose second element is `n`, whose first element is less than or equal to `m`, and whose third element is smaller than `b`. (Observe that we switch the order of `n` and `mN`, in order to obtain triples that are ordered ascendingly.)

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

Fixpoint inner_cycle (n:positive) (m:nat) (b:positive) : target_list :=
  match m with
  | O ⇒ 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.
```

Afterwards, in function `outer_cycle` below, we recur on `n`, calling `inner_cycle` with `n` as first and second argument. This yields a list containing all Pythagorean triples whose first two elements are smaller than or equal to `n` (in ascending order) and whose third element is smaller than b. By calling this function with both parameters instantiated with n, we obtain the list of all Pythagorean triples with all elements smaller than n.

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

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

In the next step, we map this list into a list of Clause by generating two Clauses 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) :=
  match t with
  | nil ⇒ nil
  | tuple::t' ⇒ (map pos tuple) :: (map neg tuple) :: (target_formula t')
  end.
```

Finally, we make this into an object of type ICNF. Function make_BPT_list (omitted) pairs each clause with an identifier; the precise values of the identifiers is immaterial, as long as they are all distinct, so for simplicity of definition we assign natural numbers in sequence to each clause. Function make_ICNF transforms this list into a binary tree, using the index to determine the path from the root to the node where the clause is placed (see [9] for details).

```
Definition Pythagorean_formula (n:nat) :=
  make_ICNF (make_BPT_list (BPT_formula n)).
```

We now formally prove the relation between this family of formulas and existence of a coloring with the positive Pythagorean property: every such coloring is a valuation that satisfies all formulas in the family. In particular, if one of these formulas is unsatisfiable, then there is no coloring of the natural numbers with the (positive) Pythagorean property. This proof explores the fact that colorings and valuations have the same type.

```
Lemma BPT_formula_sat : ∀ (C:coloring), pythagorean_pos C →
  ∀ n, satisfies C (Pythagorean_formula n).

Parameter TheN : nat.

Definition The_CNF := Pythagorean_formula TheN.

Theorem Pythagorean_Theorem : unsat The_CNF → ∀ C, ~pythagorean_pos C.
```

Due to the memory requirements of Coq, it is impossible to compute the relevant formula (with TheN set to 7826) inside the theorem prover.[2] Therefore, we use program extraction to obtain a correct-by-construction OCaml program that can compute this formula. To connect this formula with previous results [13, 9, 8] we need to change two simple things: (1) the indices in the variables in the formula need to be increased by 10,000 (so that e.g. the number 3 corresponds to variable $x_{10,003}$) and (2) the identifiers assigned to each clause need to be changed. The first item is a simple change in the formalization, which we did for experimentation

[2] This is a known limitation: Coq produces a warning message when natural numbers larger than 5000 used.

purposes but omit here (since it needs to be reverted for the simplification step that we describe in the next section). For the second item, we used the ability of the extracted checker from [8] to verify entailments by processing a sequence of SAT-solving actions and matching the resulting CNF (ignoring clause identifiers) against another one given as input. By considering the empty sequence of actions, we get a certified proof that the formula we generate entails the original formula from the proof in [13].

The running time for this experiment was just over 34 minutes (2055.39s) with a peak memory consumption of 22.67 MB. The vast majority of time was spent on generating the formula, while the entailment check was performed in less than 1 minute.

4 Simplifying the Derived Formula

We now focus on the transformation of the original formula into a simpler one. Interestingly, for proving that the Boolean Pythagorean Triples problem has no solution, there is a trivial proof that the simplification step is sound: since it only deletes clauses, if the simplified CNF is unsatisfiable, then the original one must also be unsatisfiable. However, the authors of [13] make a stronger claim: simplification preserves not only satisfiability, but also *unsatisfiability*. This property is essential for their proof that the numbers in the set $\{1,\ldots,7824\}$ *can* be colored with two colors such that no Pythagorean triple is monochromatic. We also formalize this stronger result.

In [13], the simplification step was expressed as a trace in DRAT format, and could therefore be verified by the checker from [8]. However, the underlying argument can be explained mathematically, as was done in the introduction, and is therefore suitable to a direct formalization in Coq. We formalize a more general property: if t is a list of tuples of natural numbers[3], a is a number that does not occur in any element of t, ℓ is a tuple containing a and at least one other element and C is a coloring such that no element of t is monochromatic under C, then we can find a coloring C' such that $t \cup \{\ell\}$ is monochromatic under C'.

We begin by recursively defining two predicates `no_occurrence` and `one_occurrence`, characterizing numbers that occur, respectively, in none or in exactly one tuple of a given list of tuples.

```
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 t, 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 t, one_occurrence p t →
  ∃ l,  In p l ∧ In l t ∧ (∀  l',  In l'  t → l <> l' → ~In p l').
```

[3] We use t for an element of the type `target_list` defined earlier.

Lemma no_occurrence_char reformulates the recursive definition as a global property of the list of tuples. Lemma one_occurrence_find states that we can single out the only tuple l where p occurs. From this lemma we can prove that, if that tuple is removed, then p does not occur in the result.

We are interested in colorings that do not make any element of a target_list monochromatic (predicate colorful defined below). If C is colorful with respect to list t and a does not occur in any element of t, then we can add any list containing a and at least one other element to t and find a colorful coloring C' of the result.

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

Lemma colorful_add : ∀ t a, no_occurrence a t →
  ∀ C, colorful C t → ∀ b, a <> b →
  ∀ l, In a l → In b l → ∃ C', colorful C' (l::t).
```

The proof of this lemma is by case analysis on the possible values of (C a) and (C b). If they are distinct, then we 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).

We now define the iterative simplification algorithm (function simplify below). Given a list of tuples and a list of natural numbers, we go through each element of the latter and check whether it appears in exactly one tuple (one_occurrence_dec p t); in the affirmative case, we remove the relevant tuple from the list (using function remove_number) before recurring. By induction using lemma colorful_add we show that any colorful coloring of the natural numbers with respect to the simplified list monochromatically can be used to construct a colorful coloring with respect to the original list.

```
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.

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

We now apply this construction to the family of formulas Pythagorean_formula constructed in the previous section, 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).

Definition simplified_Pythagorean_formula (n:nat) (l:list positive) :=
  make_ICNF (make_BPT_list (simplified_BPT_formula n l)).

Parameter The_List : list positive.

Definition The_Simple_CNF := simplified_Pythagorean_formula TheN The_List.
```

Particularizing 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.

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

Theorem Pythagorean_Theorem' : unsat The_Simple_CNF → ∀ C, ~pythagorean_pos C.
```

We now move on to the symmetry break, also discussed in the introduction: fixing the color of the number 2520. This is also simple to formalize in Coq. We first show that we can fix the color of a particular number – if a Pythagorean coloring of the natural numbers exists, then one must exist with the particular chosen color: either the original one, or the one obtained by flipping the color assigned to each number. Then we use this result to show that we can add any (single) unit clause to the simplified formula obtained above without losing any of the properties proved earlier.

```
Lemma fix_one_color : ∀ C, pythagorean_pos C →
  ∀ n b, ∃ C', pythagorean_pos C' ∧ C' n = b.

Parameter TheBreak : positive.

Definition The_Asymmetric_CNF :=
  make_ICNF (make_BPT_list
    ((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, ~pythagorean_pos C.
```

Again, we can use program extraction to compute this last formula from a correct-by-construction OCaml program. This requires constructing the list of numbers that are used to eliminate triples in the simplification procedure, which is done by an untrusted piece of code from the trace of the original simplification proof. (Since all lemmas are universally quantified on this list, soundness of the result is not affected by errors in this untrusted code – although such errors would not generate the formula from [13].) This time, the only difference between the CNF generated from the Coq formalization and the one produced in [13] lies in the identifiers assigned to each clause, since the original simplification procedure introduced new variables whose indices coincide with the natural numbers they correspond to. Therefore, we again use the extracted checker described in [8] to prove that this new formula entails the old one, allowing us to reuse the unsatisfiability results proved in previous work.

The running time for this experiment was just over 35 minutes (2125.98s) with a peak memory consumption of 15.8 MB. The vast majority of time was spent on generating the formula, while the entailment check was performed in less than 1 minute.

5 Cube-and-Conquer

To formalize the remaining steps in Figure 1, we need to focus on the methodology of cube-and-conquer [14]. The idea behind this methodology is simple: instead of looking for a satisfying assignment for a particular formula, consider its conjunctions with different sets of literals (the cubes) such that every possible assignment satisfies one of the possible cubes. For example, if φ is a formula on two variables x and y, the cubes could be $\{x\}$, $\{\bar{x}, y\}$ and $\{\bar{x}, \bar{y}\}$, and instead of trying to satisfy φ we would check the three formulas $\varphi \wedge x$, $\varphi \wedge \bar{x} \wedge y$ and $\varphi \wedge \bar{x} \wedge \bar{y}$. In order to ensure the side condition (every assignment satisfies one of the cubes) we would also need to check that the formula $\bar{x} \wedge (x \vee \bar{y}) \wedge (x \vee y)$, defined as the conjunction of the disjunction of the literals in each cube, is unsatisfiable.

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. When checking proofs, however, this complex step is avoided – the cubes are given as part of the input.

We formalize this process in Coq as follows. A `Cube` is simply a list of literals; function `Cubed_CNF` joins a cube with a `CNF`, adding the former's literals one by one. (Recall that a `CNF` is a binary tree of clauses.) Function `noCube` takes a list of cubes and builds the `CNF` corresponding to the negation of their disjunction; function `negate` implements negation on literals (i.e., it changes (`pos n`) to (`neg n`) and conversely).

```
Definition Cube := list Literal.

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

Fixpoint neg_cube (C:Cube) : Clause :=
  match C with
  | nil ⇒ nil
  | l ::  c ⇒ negate l :: (neg_cube c)
  end.

Fixpoint noCube (C:list Cube) : CNF :=
  match C with
  | nil ⇒ nought
  | (c ::  C') ⇒ CNF_add (neg_cube c) (noCube C')
  end.
```

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 Cubes,
  (∀  c, In c Cubes → unsat (Cubed_CNF Formula c)) → unsat (noCube Cubes) → unsat Formula.
```

This result cannot be directly combined with our previous development, however, as the type of the CNFs in this lemma is different from the type of the formula `The_Asymmetric_CNF` generated earlier. Therefore, we define a conversion function from `CNF`s to `ICNF`s by giving each clause a number, similar to `make_BPT_list` in the previous section. This function is called

CNF_to_ICNF, and we omit its definition. The key result is that it does not change satisfiability; the direction in which we use this result is expressed in the following lemma.

```
Lemma CNF_to_ICNF_unsat : ∀ c, unsat (CNF_to_ICNF c) → unsat c.
```

We then proceed to checking the application of cube-and-conquer to The_Asymmetric_CNF. Again we extract the relevant functions to OCaml and rely on the soundness of program extraction.

First, we build the list of cubes in OCaml (using untrusted code to process the files generated in [13]) and pass it as argument to noCube; using the extracted function from the lemma that states that equality of CNFs is decidable, we check that this generates exactly the same CNF as the one used in the proof of unsatisfiability in [13], which was checked in [9].[4] This ensures that the second hypothesis in Lemma CubeAndConquer_lemma holds. This step required 88.67s of CPU time and a peak memory consumption of 3.16 GB.

Next, using exactly the same code, we build the same list of cubes and pick from it each individual element; we pass this, together with The_Asymmetric_CNF, as arguments to Cubed_CNF, transform it into an ICNF, and show that this entails the corresponding cubed formula from [13], whose proof of unsatisfiability was again formally verified by the checker in [9]. By Lemma CNF_to_ICNF_unsat this implies that the CNF in the corresponding premise of Lemma CubeAndConquer_lemma – namely, The_Asymmetric_CNF – also holds. As a consequence, The_Asymmetric_CNF is unsatisfiable.

This last step required running 1000 jobs checking 1000 cubes each on 40 nodes, running 25 jobs in parallel on each node. Each of these jobs took between 45,859.65s and 51,188.46s (average 49,440.38s and median 49,590.77s) to complete. Within each job, approximately 2100s were spent on generating The_Asymmetric_CNF, and the remaining time on generating the 1000 formulas using Cubed_CNF and checking that they entail the corresponding formula checked in [9]. The total CPU time used amounts to just over 1.5 CPU years, and the peak memory consumption per node was 41.46 GB.

The only part in this proof that are not formally verified is the chaining of arguments at the meta-level: the applications of lemmas CNF_to_ICNF_unsat and CubeAndConquer_lemma. In particular, we need to trust that the implementation does test all cubes in the list of cubes: this is done by untrusted code, and not by iteration over the list using extracted code. Although in principle it would have been possible to do this step formally, the time requirements made it unpractical (the verification was run in parallel).

6 Conclusions

The main contribution of this article is a formalization of the Boolean Pythagorean Triples problem in the theorem prover Coq, together with a strategy for generating a family of propositional formulas whose unsatisfiability implies that the Pythagorean equation is partition regular. We choose the particular formula whose unsatisfiability we know how to prove (using proof witnesses from [13]), simplify it, again by certified methods, and split the proof of its unsatisfiability in one million (plus one) subproblems. The correspondence between unsatisfiability of the original formula and unsatisfiability of the 1,000,001 derived formulas is a consequence of the formalization, in Coq, of the soundness of the cube-and-conquer methodology.

The soundness of our development depends on previous work only as regards the ability to check proofs of unsatisfiability by a correct-by-construction verifier described in [9]. We do *not*

[4] The checker extracted in the latter development actually builds a CNF as an intermediate step, which allowed us to bypass constructing an ICNF from it.

depend on the results from [13]: although we (intensively!) use the data from their experiments as oracle to our extracted code, that data is used in an *untrusted* manner, meaning that it is checked for correctness before being used. This is illustrated by the universal quantifications in all soundness results in our formalization (namely, `Pythagorean_Theorem''`, which quantifies over the parameters `TheN`, `The_List` and `TheBreak`, and `CubeAndConquer_lemma`, which quantifies over the an arbitraty list of cubes). Likewise, we do *not* depend on the results from [8] in an essential manner: all the unsatisfiability proofs we consider use the GRIT format already verifiable with the checker from [9]. Although the entailment check capability was added in that work, we only use it with an empty number of inferences, so that all we are testing is set equality of binary trees.

Our formalization relies on meta-level arguments in two places.

1. The formulas generated by the cube-and-conquer methodology are shown to entail formulas that were previously shown to be unsatisfiable. In this step, the old formulas are provided as arguments to two different functions: the entailment checker and the unsatisfiability checker. We guarantee soundness of this step by using *exactly* the same untrusted code to construct the formulas from *exactly* the same input files.

2. The final entailment (arrow (c) in Figure 1) requires connecting the 1,000,001 proofs of unsatisfiability. Since these were done independently and in a parallel way, we argue that `CubeAndConquer_lemma` can in principle be applied, since the formulas generated and tested were obtained by applying exactly the same functions as in the premises of that lemma. However, one needs to trust that we indeed covered *all* the cases in the list of cubes, since this step was done by manually written (untrusted) code.

These dependencies arise for the need to parallelize code and, on a lower scale, due to the enormous computational requirements for verifying the 1,000,001 unsatisfiability proofs, which led us to try to reuse previous results. It would not be hard to extend our formalization to a checker that would, theoretically, combine all the steps. Such a checker would receive as arguments also the oracles for the 1,000,001 unsatisfiability proofs, and iterate through the list of cubes to generate all the formulas and check their unsatisfiability. Its soundness could then be stated and proved within Coq, by applying `CubeAndConquer_lemma`. However, since Coq is currently unable to produce parallelizable code, running this code would require several years on state-of-the-art hardware, which is clearly unfeasible.

References

[1] Eyad Alkassar, Sascha Böhme, Kurt Mehlhorn, and Christine Rizkallah. A framework for the verification of certifying computations. *J. Autom. Reasoning*, 52(3):241–273, 2014.

[2] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development.* Texts in Theoretical Computer Science. Springer, 2004.

[3] Manuel Blum and Sampath Kannan. Designing programs that check their work. In David S. Johnson, editor, *STOC*, pages 86–97. ACM, 1989.

[4] Michael Codish, Luís Cruz-Filipe, Michael Frank, and Peter Schneider-Kamp. Sorting nine inputs requires twenty-five comparisons. *Journal of Computer and System Sciences*, 82(3):551–563, 2016.

[5] Michael Codish, Michael Frank, Avraham Itzhakov, and Alice Miller. Computing the Ramsey number $R(4,3,3)$ using abstraction and symmetry breaking. *Constraints*, 21(3):375–393, 2016.

[6] Joshua Cooper and Ralph Overstreet. Coloring so that no pythagorean triple is monochromatic. *CoRR*, abs/1505.02222, 2015.

[7] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[8] Luís Cruz-Filipe, Marijn Heule, Warren Hunt, Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. *CoRR*, abs/1610.06984, 2016.

[9] Luís Cruz-Filipe, Joao Marques-Silva, and Peter Schneider-Kamp. Efficient certified resolution proof checking. *CoRR*, abs/1610.06984, 2016.

[10] Luís Cruz-Filipe and Peter Schneider-Kamp. Checking the Boolean Pythagorean Triples conjecture. Available from: `http://imada.sdu.dk/~petersk/bpt/`.

[11] Luís Cruz-Filipe and Peter Schneider-Kamp. Formalizing size-optimal sorting networks: Extracting a certified proof checker. In C. Urban and X. Zhang, editors, *ITP*, volume 9236 of *LNCS*, pages 154–169. Springer, 2015.

[12] Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In Maria Paola Bonacina, editor, *CADE-24*, volume 7898 of *LNCS*, pages 345–359. Springer, 2013.

[13] Marijn Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In Nadia Creignou and Daniel Le Berre, editors, *SAT*, volume 9710 of *LNCS*, pages 228–245. Springer, 2016.

[14] Marijn Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In Kerstin Eder, João Lourenço, and Onn Shehory, editors, *HVC 2011*, volume 7261 of *LNCS*, pages 50–65. Springer, 2012.

[15] Boris Konev and Alexei Lisitsa. Computer-aided proof of Erdős discrepancy properties. *Artif. Intell.*, 224:103–118, 2015.

[16] Bruce M. Landman and Aaron Robertson. *Ramsey Theory on the Integers*, volume 24 of *The Student Mathematical Library*. AMS, 2004.

[17] P. Letouzey. Extraction in Coq: An overview. In A. Beckmann, C. Dimitracopoulos, and B. Löwe, editors, *CiE 2008*, volume 5028 of *LNCS*, pages 359–369. Springer, 2008.

[18] Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Computer Science Review*, 5(2):119–161, 2011.

[19] Freek Wiedijk, editor. *The Seventeen Provers of the World*, volume 3600 of *LNCS*. Springer, 2006.