Formal Methods in Computer-Aided Design 2023

Verified Encodings for SAT Solvers

Cayden R. Codel
Computer Science Department
Carnegie Mellon University
Pittsburgh, USA
ccodel@cs.cmu.edu

Jeremy Avigad
Department of Philosophy
Carnegie Mellon University
Pittsburgh, USA
avigad@cmu.edu

Marijn J. H. Heule
Computer Science Department
Carnegie Mellon University
Pittsburgh, USA
marijn@cmu.edu

***Abstract*—Satisfiability (SAT) solvers are versatile tools that can solve a wide array of problems, and the models and proofs of unsatisfiability emitted by SAT solvers can be checked by verified software. In this way, the SAT toolchain is trustworthy. However, many applications are not expressed natively in SAT and must instead be *encoded* into SAT. These encodings are often subtle, and implementations are error-prone. Formal correctness proofs are needed to ensure that implementations are bug-free.**

In this paper, we present a library for formally verifying SAT encodings, written using the Lean interactive theorem prover. Our library currently contains verified encodings for the parity, at-most-one, and at-most-k constraints. It also contains methods of generating fresh variable names and combining sub-encodings to form more complex ones, such as one for encoding a valid Sudoku board. The proofs in our library are general, and so this library serves as a basis for future encoding efforts.

I. Introduction

Satisfiability (SAT) solvers are powerful and versatile tools. They solve hardware and software verification tasks [1, 2], they are used in satisfiability modulo theory solvers [3, 4], and they are instrumental in resolving longstanding open problems in mathematics [5, 6]. Impressed by their strength and utility, Donald Knuth called SAT solvers a “killer app” [7].

Modern SAT solvers are also *trustworthy*. Since the SAT problem is in NP [8], models can be efficiently checked. When no model exists, solvers emit a certificate of unsatisfiability, which is a step-by-step proof written in a formal proof system [9, 10], the de facto standard being DRAT [11]. Proof checkers can then check that the certificate is correct [12, 13]. Proof checkers are simple pieces of software; several checkers have been formally verified [14–17].

Many applications are not expressed natively in SAT and must instead be *encoded* into SAT. The challenge is that encodings are often subtle, and so it is easy to make off-by-one errors and other bugs when implementing an encoding. Detecting errors in an encoded formula is even more challenging: in extreme cases, encodings can contain hundreds of thousands of variables and clauses. A single wrong variable or clause can render the entire formula useless. One way to get rid of bugs in encodings is to verify them in a proof assistant.

In this paper, we present a library for verifying SAT encodings, written with the proof assistant Lean 3 [18]. So far, we have verified encodings for the parity, at-most-one, and at-most-k constraints. These encodings are common and are used in applications such as cryptography [19, 20], haplotype inference [21], and approximate model counting [22].

In addition to our correctness proofs, we discuss the techniques we developed in our library. One major contribution in our library is a way of introducing and managing fresh variables, which are commonly used to minimize the number of clauses in an encoding. Another contribution is a method of composing constraints and encodings together to form more complex ones while keeping correctness proofs short. We demonstrate how these operations are used in an encoding of Sudoku in Section VI.

II. Preliminaries

Boolean variables range over the classical truth values true ($\top$) and false ($\bot$). *Boolean literals* are positive or negative forms of boolean variables, written as x and $\overline{x}$, respectively. *Truth assignments* τ give truth values to sets of boolean variables. When $\tau(x) = \top$, then $\tau(\overline{x}) = \bot$. If F is a propositional formula that evaluates to true under τ, written as $\tau(F) = \top$, then we say that τ *satisfies* F. If there exists a τ that satisfies F, then F is *satisfiable*.

Let $\mathrm{vars}(\cdot)$ be the set of variables contained in a propositional formula. We overload $\mathrm{vars}(\cdot)$ for τ to mean the set of variables that τ is defined over. If τ and τ' are truth assignments such that $\mathrm{vars}(\tau) \subseteq \mathrm{vars}(\tau')$ and $\tau(x) = \tau'(x)$ whenever $x \in \mathrm{vars}(\tau)$, then τ' *extends* τ.

Most modern SAT solvers only accept formulas in *conjunctive normal form* (CNF). A formula is in CNF if it is a conjunction of clauses, with each clause a disjunction of literals. Unless otherwise noted, when we refer to a formula F, we assume it is in CNF.

Any problem may admit many encodings. From a mathematical point of view, the choice of encoding doesn’t matter as long as each is correct, but in practice, solvers perform better on encodings with fewer variables and clauses [23, 24]. Generally, compact encodings introduce fresh variables to reduce the overall number of clauses, but at the cost of added complexity.

In this paper, we focus on encodings of n-ary boolean constraints. Let $X = x_1, \ldots, x_n$ represent the inputs to an n-ary boolean constraint C, and let F be any propositional formula. When $\mathrm{vars}(F) \subseteq X$, then F encodes C if and only if it *defines* it: for every full assignment τ on X, we require

$$C(\tau(x_1), \ldots, \tau(x_n)) \leftrightarrow \tau(F) = \top.$$

This work was partially supported by the Hoskinson Center for Formalized Mathematics and by NSF grant CCF-2108521.

https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_22

Yet F may use additional variables. The following definition handles the more general case.

Definition 1 (Encoding a boolean constraint): Let C be a boolean constraint, F be any propositional logic formula, and $X = x_1, \ldots, x_n$ be variables representing the inputs to C. Then F *encodes* C if and only if: for every assignment τ on X, $C(\tau(x_1), \ldots, \tau(x_n))$ if and only if F is satisfied by some assignment that extends τ.

Using the language of quantified propositional logic, this amounts to saying that C is defined by $\exists y_1, \ldots, y_m\ F$, where the y_i are the additional variables appearing in F. It may help to think of the y_i as auxiliary objects that are required to satisfy their descriptions in F.

In our library, an *encoding function* for a constraint C takes an input list of boolean literals and returns an encoding for C on those literals.[1] An encoding function is *correct* for C if the formulas it produces encode C on all valid inputs. We will see in Section IV that this notion of correctness will need to be augmented to account for fresh variable generation.

III. The constraints and their encodings

We now discuss the constraints and encodings that appear in our proof library and develop the intuition for why the encodings are correct. These intuitions form the basis for the correctness proofs presented in Section V.

A. The parity constraint

The n-ary *parity constraint* is encountered in problems from a wide range of domains, such as cryptography [19, 20], approximate model counting [22], the creation of matrix multiplication schemes [25, 26], and the construction of set membership filters [27]. Many encodings for this constraint have been proposed [26, 28–30], and to the best of our knowledge, these encodings remain unverified. In our library, we prove the correctness of two particular encodings: the direct encoding and a recursive encoding.

The parity constraint concerns the true-false parity of a set of boolean variables. Let $\textsc{parity}(X)$ be satisfied iff an odd number of the x_i are true. One way to write $\textsc{parity}$ in propositional logic is with the XOR ($\oplus$) connective, where $x \oplus y = \top$ iff exactly one of x and y are true. We can thus write $\textsc{parity}(X)$ as $x_1 \oplus \cdots \oplus x_n$.

The first encoding we examine is the *direct* (or *naive*) *encoding*. Every boolean constraint has a direct encoding that is essentially a spelled-out truth table. Direct encodings are sometimes chosen because they are simple to implement, since they don't introduce fresh variables, but they often produce formulas with many clauses, and so they are not preferred on large inputs. In the case of the parity constraint, its direct encoding produces a formula with 2^{n-1} clauses. Such a formula quickly becomes intractable for solvers.

[1]It is convenient for encoding functions to accept lists of *literals* as input rather than lists of variables since that allows the inputs to be negated. This is useful when implementing several encodings, especially recursive ones.

Definition 2 (Direct encoding of $\textsc{parity}$*):* The direct encoding of $\textsc{parity}$ on boolean literals $X = x_1, \ldots, x_n$ is

$$\textsc{direct}_{\textsc{parity}}(X) = \bigwedge_{\text{even \# of negations}} \left(\bigvee_{i=1}^{n} \pm x_i \right).$$

To see how the encoding works, consider any assignment τ that does not satisfy $\textsc{parity}$. We know by the definition of $\textsc{parity}$ that an even number of the x_i must be true under τ. Since the direct encoding includes every clause with an even number of negations, we can find the clause that negates exactly those x_i that are true under τ. That clause evaluates to false under τ. Thus, the only truth assignments that satisfy every clause are those that set an odd number of the x_i to true, which are precisely the assignments that satisfy $\textsc{parity}$.

The second encoding is a recursive one loosely based on the Tseitin transformation [31]. The Tseitin transformation takes a propositional logic formula F and produces an equisatisfiable CNF formula that has length linear in the size of F by recursively introducing fresh literals via if-and-only-if relations with sub-formulas. This method of introducing fresh variables is used in the recursive encoding.

We first fix a *cutting number* $k \geq 3$ to determine how to split $x_1 \oplus \cdots \oplus x_n$ into two sub-constraints. We then replace the first $k-1$ literals with a fresh literal and recurse:

$$\begin{aligned} R_k(X) &= (\textsc{parity}(X_{[1,k)}) \leftrightarrow y) \wedge (y \oplus R_k(X_{[k,n]})) \\ &= \textsc{direct}_{\textsc{parity}}(X_{[1,k)}, \overline{y}) \wedge R_k(y, X_{[k,n]}), \end{aligned}$$

where we get the second line by rearranging the variables (recall that $a \leftrightarrow b$ is equivalent to $a \oplus \overline{b}$) and by replacing the left instance of the parity constraint with the direct encoding.

Note that because $\oplus$ is commutative, we have a choice of where to place y in the recursive step. In practice, it is common to place y in either the leftmost or rightmost position. Encodings that use the former method are called *linear*; the latter, *pooled*. We prove the more general result that the encoding is correct for any permutation of $x_k, \ldots, x_n$ and y.

The choice of where to place y in the recursive transformation can have a big impact on solver performance. For example, consider an assignment that satisfies the parity constraint but falsifies the two clauses containing the literals x_1 and x_n in both encodings. The linear encoding requires $O(n)$ updates to its fresh literals to make the formula evaluate to true, while the pooled encoding only requires $O(\log n)$ updates [26].

The choice of cutting number is also critical for solver performance. When k is larger, each encoding introduces fewer fresh variables but at a cost of larger direct encoding sub-formulas. Applications are known for which cutting numbers of $k = 4$, 6, and 7 are optimal [22, 26, 28]. In our correctness proof, the cutting number is arbitrary.

Definition 3 (Recursive encoding for $\textsc{parity}$*):* Fix $k \geq 3$, and let p be a function that permutes lists. Then the recursive encoding for the $\textsc{parity}$ on literals $X = x_1, \ldots, x_n$ is

$$R_k(X) = \textsc{direct}_{\textsc{parity}}(X_{[1,k)}, \overline{y}) \ \wedge\ R_k(p(y, X_{[k,n]})),$$

where y is fresh. When $n \leq k$, the direct encoding is used instead. The linear encoding places y in the leftmost position, while the pooled encoding places y in the rightmost position.

Both encodings presented in this section encode the *positive* form of the parity constraint. To encode its negation, which is satisfied iff an even number of the x_i are true, one can either encode $\textsc{parity}(\overline{x}_1, x_2, \ldots, x_n)$ or introduce a fresh variable z and add a unit clause to ensure that z is set to true in any satisfying assignment: $z \wedge \textsc{parity}(z, x_1, \ldots, x_n)$.

B. The at-most-one constraint

Pseudo-boolean constraints appear in many applications for the SAT and maximum satisfiability problems, from scheduling to haplotype inference [21, 32–35]. One important class of pseudo-boolean constraint is the *cardinality constraint*, which can be written as

$$\sum_{i=1}^{n} a_i x_i \leq k \qquad \text{or} \qquad \sum_{i=1}^{n} a_i x_i \geq k,$$

where k is a fixed constant, $a_i \in \{\pm 1\}$, and $x_i = 1$ when $\tau(x_i) = \top$ and $x_i = 0$ otherwise. In our library, we assume that all $a_i = 1$, but we allow writing cardinality constraints in terms of boolean literals, so the two systems are equivalent.

The *at-most-one constraint* (AMO) is an especially common cardinality constraint. As its name indicates, it specifies that at most one of the x_i can evaluate to true. Many AMO encodings have been proposed [36–40]. In our library, we prove the correctness of the direct and sequential counter encodings.

Like for PARITY, the direct encoding for AMO is its CNF definition. It consists of binary clauses $(\overline{x}_i \vee \overline{x}_j)$ that disallow truth assignments that set both x_i and x_j to true. The encoding produces $\binom{n}{2} \in O(n^2)$ clauses and uses no fresh variables.

Definition 4 (Direct encoding of AMO*):* The direct encoding of the AMO constraint on boolean literals $X = x_1, \ldots, x_n$ is

$$\textsc{direct}_{\textsc{amo}}(X) = \bigwedge_{1 \leq i < j \leq n} (\overline{x}_i \vee \overline{x}_j).$$

The *sequential counter encoding* [40] is a popular linear-sized encoding. It produces $3n - 4 \in O(n)$ clauses and introduces $n - 1$ *signal variables* that propagate the truth value of any true x_i to other signal variables to ensure that all later x_j remain false. Figure 1 shows the encoding under a satisfying truth assignment.

Definition 5 (The sequential counter AMO *encoding):* The sequential counter encoding for the AMO constraint on boolean literals $X = x_1, \ldots, x_n$ is

$$\textsc{sc}(X) = \bigwedge_{i=1}^{n-1} \Big((\overline{x}_i \vee s_i) \wedge (\overline{s}_i \vee s_{i+1}) \wedge (\overline{s}_i \vee \overline{x}_{i+1}) \Big),$$

where the s_i are fresh and pairwise distinct.

In our library, we omit the clause $(\overline{s}_{n-1} \vee s_n)$ because s_n doesn't appear in any other clause. Omitting the clause keeps the number of signal variables at $n - 1$.

There are three kinds of clauses in the encoding. They are logically equivalent to

$$(x_i \rightarrow s_i) \quad \wedge \quad (s_i \rightarrow s_{i+1}) \quad \wedge \quad (s_i \rightarrow \overline{x}_{i+1}).$$

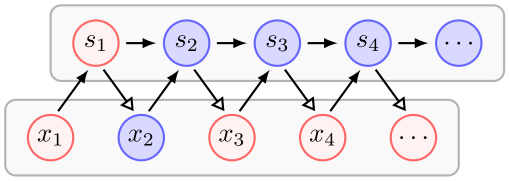

Fig. 1. The sequential counter AMO encoding under a satisfying truth assignment. Blue means the literal is true and red means the literal is false. The hollow arrow heads indicate a negated implication. Notice how the signal variables propagate that x_2 is true, enforcing that all later x_i must be false.

Writing the clauses like this makes it easier to see how the encoding works. A true x_i sets all following signal variables to true, which then forces all following x_j to false.

*C. The at-most-*k *constraint*

The sequential counter encoding can be generalized into an encoding of the *at-most-k constraint* (AMK). It introduces a $(k + 1) \times n$ matrix of signal variables and produces $O(nk)$ clauses. Clauses similar to those in the AMO encoding ensure that the matrix tracks the cumulative number of the x_i that are true. A unit clause containing the last signal variable disallows truth assignments that set more than k of the x_i to true.

Let $s_{i,j}$ be the signal variable on the ith row and jth column of the matrix. The encoding ensures that $s_{i,j}$ is set to true when at least i of $x_1, \ldots, x_j$ are true. One can think of j as defining the $X_{[1,j]}$ sub-array and i as the truth counter.

Definition 6 (The sequential counter AMK *encoding):* Let $k \geq 2$ be given. The sequential counter AMK encoding on literals $X = x_1, \ldots, x_n$ is

$$\textsc{sc}_k(X) = \left(\bigwedge_{j=1}^{n} (\overline{x}_j \vee s_{1,j}) \right) \wedge \left(\bigwedge_{i=1}^{k+1} \bigwedge_{j=1}^{n-1} (\overline{s}_{i,j} \vee s_{i,j+1}) \right)$$
$$\wedge \left(\bigwedge_{i=1}^{k} \bigwedge_{j=1}^{n-1} (\overline{x}_{j+1} \vee \overline{s}_{i,j} \vee s_{i+1,j+1}) \right) \wedge \overline{s}_{k+1,n},$$

where the $s_{i,j}$ are fresh and pairwise distinct.

There are three types of clauses in the encoding. The first two appear in the AMO encoding. The third kind, the ternary clause, is logically equivalent to $(x_{j+1} \wedge s_{i,j}) \rightarrow s_{i+1,j+1}$. Whenever $s_{i,j}$ is true, meaning that at least i of $x_1, \ldots, x_j$ are true, and x_{j+1} is true, then $s_{i+1,j+1}$ is set to true. In other words, the ternary clause propagates the truth counter up a row in the signal variable matrix when a new x_{j+1} is set to true.

Figure 2 shows the encoding under a satisfying truth assignment.

IV. Library for Verified Encodings

In this section, we present our library for verifying SAT encodings. We used the interactive theorem prover Lean 3 [18] (hereafter called Lean), and our library depends on Lean's community proof library `mathlib` [41]. Our library is open-source, and all proofs and compilation instructions can be found at the following URL:

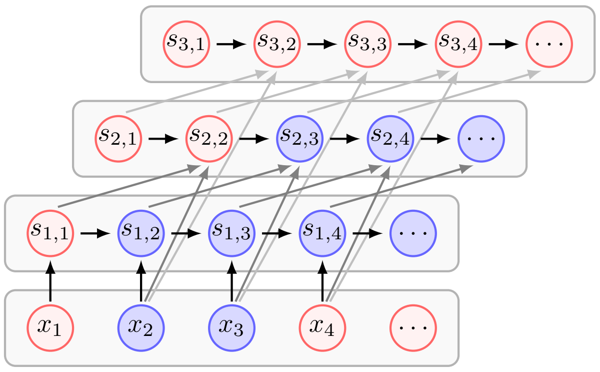

Fig. 2. The sequential counter AMK ($k = 2$) encoding under a satisfying truth assignment. Blue means that the literal is true and red means the literal is false. Notice how x_3 being true sets a new row in the signal variable matrix to true. A unit clause containing the top-rightmost signal variable disallows any assignment that sets the top row of the matrix to true.

https://github.com/ccodel/verified-encodings.

Lean's axiomatic foundation is a dependent type theory with inductive types and a type of propositions. While Lean's core logic is constructive, proofs in `mathlib` make use of classical logic. Our proofs do not depend on the specifics of Lean beyond basic facts on natural numbers, functions, lists, and sets.

Variable and theorem names in this paper may differ from those in our library since they can be verbose or cryptic. This is due to our following Lean's naming convention. While we have edited names for readability, we have hyperlinked definitions and theorems to their counterparts in our library.

A. *Library preliminaries*

We start our tour of the proof library by covering the basic objects and operations we use. Most definitions are natural and correspond to general intuition about CNF formulas, clauses, and truth assignments and the interactions between them.

To avoid using a specific type for boolean variables, we use an arbitrary type `V` that is equipped with a computable test for equality. Literals are a sum type that are either positive (`Pos v`) or negative (`Neg v`). Truth assignments are functions from `V` to `bool`. Clauses and CNF formulas are represented by lists of literals and clauses, respectively.

We define operations on these types, such as evaluation and vars($\cdot$), and we prove theorems about those operations. All the operations in our library are computable, meaning that Lean can execute them on explicit instances of clauses and formulas. As an example, consider the evaluation of clauses under a truth assignment τ and the statement that a clause evaluates to true under τ it has some literal that evaluates to true under τ.

```
def eval (τ : assignment V) (c : clause V) :=
    c.foldr (λ l b, b || l.eval τ) ff
theorem clause_eval_tt_iff {τ} {c} :
  c.eval τ = tt ↔ ∃ l ∈ c, l.eval τ = tt
```

The `foldr` function folds a binary operation over the elements of a list, and `l.eval τ` evaluates literal `l` under τ. True and false in Lean are written as `tt` and `ff`, respectively.

(The expressions `c.foldr` and `l.eval` are in Lean's *anonymous projection notation*. Because Lean infers `l` to have type `literal`, it interprets `l.eval τ` as `literal.eval l τ`, inserting `l` as the first explicit argument of the correct type. Similarly, because the type `clause V` reduces to `list V`, Lean interprets `c.foldr` as `list.foldr c`. We use this notation often.)

One consequence of our decision to represent truth assignments as maps from `V` to `bool` is that any assignment in our library is a *full* assignment. However, we have defined in this paper that assignments are (potentially) *partial* maps on sets of variables. Having assignments be full maps makes it easier to construct and combine them, but it adds a small amount of overhead in correctness proofs to manage the sets on which the assignments are "defined." It also requires us to modify some definitions in Section II. For example, instead of saying that τ_2 extends τ_1, we say that τ_2 agrees with τ_1 on a specified set of variables V (`agree_on`). Thus, when τ_1 and τ_2 agree on the variables in a clause, a formula, etc., evaluation and other operations are equivalent under the two assignments.

A common pattern in our library is to start with an assignment τ_1 that satisfies a property on a set of variables V, and then "extend" it to a new assignment by setting explicit truth values for variables not in V. One way to construct such assignments is to use `aite` (short for "assignment if-then-else"). Then, as long as the object under consideration only has variables in (or not in) V, the `aite` assignment can be reduced back to one of τ_1 or τ_2.

```
def aite (V : finset V) (τ₁ τ₂) :=
  λ v, if v ∈ V then τ₁ v else τ₂ v
theorem aite_pos {V} {v} :
  v ∈ V → ∀ τ₁ τ₂, (aite V τ₁ τ₂) v = τ₁ v
```

B. *Fresh variable generation and management*

Almost all compact SAT encodings introduce *auxiliary* or *fresh variables*, which are variables that don't appear in the input. For mathematicians (and most computer scientists), generating fresh variables is easy: one assumes that there exists a set with enough fresh variables and that these variables can be chosen at will. But we have no such *a priori* assumption when we use Lean. So, we took inspiration from de Bruijn indices [42] and `gensym` objects [43] (such as in Lisp) to create our own `gensym` object that generates fresh variables.

In our library, a `gensym` object is a pointer n on the natural number line and an injective function $f : \mathbb{N} \to \alpha$ for α an arbitrary type. The `gensym`'s pointer starts by default at $n = 0$, but it can be initialized to a higher value, perhaps to avoid variables already present in a formula. The `fresh` operation provides a fresh variable under f and an updated `gensym` with an incremented pointer. Batches of fresh variables can be acquired with `nfresh`. Because f is injective, the generated variables are all distinct.

A useful notion for a `gensym` is its *stock*, the set of variables the `gensym` can produce. The stock S of a `gensym` g with offset n is $S(g) := \{x : \alpha \mid \exists\, d \in \mathbb{N},\ f(n + d) = x\}$.

To ease proof burdens when proving encodings correct, we provide lemmas that state how an updated gensym's stock and generated fresh variables relate to the original stock. For example, here are two lemmas we use often.

```
lemma fresh_stock_subset (g : gensym V) :
  g.fresh.2.stock ⊆ g.stock
lemma fresh_not_mem_fresh_stock (g) :
  g.fresh.1 ∉ g.fresh.2.stock
```

The fresh operation returns a pair of a variable and an updated gensym object. In Lean, the components of a pair are accessed through the .1 and .2 notation, which are abbreviations for .fst and .snd.

Sometimes it is more convenient to index into a gensym's stock rather than request fresh variables. The nth operation takes a number i and returns the fresh variable that would have been generated after i calls to fresh, but without updating the gensym. Using nth makes proving correctness more challenging, however, since many lemmas associated with fresh cannot be applied to nth, so fresh is the recommended operation.

An equivalent definition for gensym is to only manage the injective function f, where calls to fresh would give back $f := (\lambda n, f(n+1))$. We chose the offset representation because it is easier to reason about natural numbers in Lean than anonymous lambda functions. For example, it is easy to prove that g.nth i ≠ g.nth j when i ≠ j due to the injectivity of f, whereas the proof for the alternate definition would be a more roundabout induction proof.

C. Encodings and correctness

We now have enough tools and machinery at hand to discuss how the encodings and their proofs of correctness appear in our library. We start by defining when a formula encodes a constraint. In our library, a constraint is a function from list bool to bool. Using agree_on in place of assignment extension, we represent Definition 1 like so:

```
def encodes (C) (F) (l : list (literal V)) :=
∀ τ, (C.eval τ l = tt) ↔ ∃ σ, F.eval σ = tt
    ∧ (agree_on τ σ (vars l))
```

Encoding functions take lists of literals to CNF formulas. In our library, we require a gensym to generate fresh variables, so we add the gensym as an explicit input and output.

```
def enc_fn (V : Type*) := list (literal V) →
     gensym V → cnf V × gensym V
```

The definition of correctness follows naturally from the one in Section II: that for any input list of literals l, the resulting formula produced by the encoding function encodes the constraint on l. We must also add the assumption that the variables in l and the stock of the provided gensym are disjoint, ensuring that the fresh variables are actually fresh.

```
def is_correct (C) (e : enc_fn V) :=
  ∀ ⦃l⦄ ⦃g⦄, disjoint (vars l) g.stock →
    encodes C (e l g).1 l
```

Often, proving correctness is insufficient. Many encodings are comprised of sub-encodings, and to prove that these composite encodings are correct, we need to know that the sub-encoding functions "play nice" with the gensym object as it passes from one to the next. Otherwise, the combination of two encoding functions may result in unexpected behavior. For example, fresh variables could be taken from the gensym's stock without updating the gensym, leading to variable clash when sub-formulas are combined.

To solve this problem, we introduce a notion of well-behavedness. Intuitively, an encoding function is *well-behaved* if the variables of its formulas either come from its input list l or from the gensym's stock, and its output gensym is updated to avoid those fresh variables. All the encodings in this paper are well-behaved.

```
def is_wb (e : enc_fn V) := ∀ ⦃l⦄ ⦃g⦄,
  disjoint (vars l) g.stock →
  (e l g).2.stock ⊆ g.stock ∧
  (e l g).1.vars ⊆ (vars l) ∪
    (g.stock \ (e l g).2.stock)
```

Well-behaved encoding functions can be combined together safely. We define an append operation, written as ++, to run one encoding function and then the other on the same list of input literals. If each well-behaved encoding function encodes a constraint, then their combination is also well-behaved and encodes the boolean-AND of the two constraints.

```
def append (e₁ e₂) : enc_fn V := λ l g,
  let ⟨F₁, g₁⟩ := e₁ l g in
  let ⟨F₂, g₂⟩ := e₂ l g₁ in ⟨F₁ ++ F₂, g₂⟩
```

We also define a fold operation that folds append over a list of encodings from left to right in the natural way. Analogous append and fold operations are defined for constraints as well, where append is the boolean-AND of the outputs.

V. Proving the Encodings Correct

In this section, we present the correctness proofs for the encodings we presented in Section III. The proofs generally follow the intuition of correctness given with the definition of the encodings, but we report challenges, quirks, or surprises.

A. The parity encodings

We represented PARITY by folding ⊕ (written as bxor in Lean) across the input. A lemma states that the constraint is satisfied iff an odd number of inputs are true.

```
def parity := λ l, l.foldr bxor ff
lemma parity_eq_bodd : parity.eval τ l =
    bodd (clause.count_tt τ l)
```

We implemented the direct encoding by adding either x_1 or its negation to each of the 2^{n-1} (ordered) clauses on $x_2, \ldots, x_n$. The proof of correctness is short (only about 30 lines) and proceeds along the already given intuition: By specifying that all clauses in the encoded formula have an even number of negations, any falsifying assignment for PARITY has a corresponding clause in the formula that it does not satisfy.

The recursive encoding and its correctness proof are more interesting. In Lean, we take a cutting number `k` and a permutation function `p` and implement the encoding recursively.

```
def recursive_parity {k} (hk : k ≥ 3) {p}
  (hp : ∀ l, perm l (p l)) : enc_fn V
| l g := if l.length ≤ k then
  direct_parity l g else
  let ⟨y, g₁⟩ := g.fresh in
  let ⟨lhd, ltl⟩ := l.split (k - 1) in
  let ⟨Frec, g₂⟩ := recursive_parity (
    p (Pos y :: ltl) ) g₁ in
  ⟨(direct_parity (lhd ++ [Neg y]) g₁).1
    ++ Frec, g₂⟩
```

The `perm` relation specifies whether two lists are permutations of each other. The `split` operation returns two halves of the list, split at the specified index.

The proof of correctness proceeds by strong induction on the input list `l`. Let τ be the truth assignment in the `encodes` judgment. The reverse direction (that if there exists an assignment σ that agrees with τ on the variables in `l` and satisfies the encoded formula, then τ satisfies PARITY) is almost trivial. Applying the correctness proof for the direct encoding and the induction hypothesis on the recursive sub-formula gives two satisfied PARITY constraints. Dropping the fresh variable y in both gives a single satisfied PARITY constraint.

The forward direction is more involved. To use the induction hypothesis, we must show that PARITY$(y, X_{[k,n]})$ is satisfied under some truth assignment. We construct such a truth assignment ν by extending τ to include a truth value for the fresh variable y. If PARITY$(X_{[k,n]})$ is satisfied under τ, then y is set to false in ν, and true otherwise. The induction hypothesis on the sub-formula returns an assignment σ that satisfies the sub-formula and that agrees with ν on $\{y\} \cup X_{[k,n]}$. Combining σ with τ on $X_{[1,k)}$ via `aite` finishes the proof.

The general takeaway is that for recursively-defined encodings, the proof of correctness proceeds by (strong) induction on the input list and requires the explicit setting of truth values for one or more variables in an extended assignment, especially among the fresh variables. Lemmas that manipulate and reduce `aite` constructions are helpful, but the management of hypotheses about set membership ultimately remains tedious.

B. The at-most encodings

We defined the AMK constraint with Lean's `list.count` operation, which counts the number of elements in a list that match a given element. The AMO constraint is `amk 1`. The at-least-k constraint ALK and the at-least-one constraint `alo` are defined analogously.

```
def amk (k : nat) := λ l, l.count tt ≤ k
```

We implemented the direct encoding for AMO as a recursive function. Because the direct encoding doesn't require any fresh variables, we defined a base function `direct_amo'` to produce the formula. The actual encoding function passes the `gensym` through untouched.

```
def direct_amo' : list (literal V) → cnf V
| []            := []
| (lit :: ls) := (ls.map (λ m,
  [lit.flip, m.flip])) ++ (direct_amo' ls)
```

The correctness proof is straightforward and proceeds by relating both the AMO constraint and the clauses in the direct encoding to the truth value of any two elements in distinct positions in the list via a `distinct` proposition we defined.

```
def distinct {α} (a₁ a₂ : α) (l : list α) :=
  ∃ (i j : nat) (Hi : i < l.length)
  (Hj : j < l.length), i < j ∧
  l.nth_le i Hi = a₁ ∧ l.nth_le j Hj = a₂
```

We now discuss the sequential counter encodings, starting with the AMO encoding. Unlike for the recursive encoding for PARITY, the AMO encoding isn't inherently recursive, but the three clauses have the same form for each i, so a recursive implementation is possible. However, a non-recursive implementation may allow for lemmas that better capture the global behavior of the signal variables. We implemented both a recursive and non-recursive encoding function in our library to compare the proof efforts of the two methods.

The non-recursive `sc_amo` uses three helper functions that each generate one of the three types of clauses. We provide one of them, `xi_to_si`, below as an example. We omit the cases where the input list has cardinality at most one.

```
def sc_amo : enc_fn V
| l g := let n := length l in
  ⟨join (map_with_index (λ idx lit,
    xi_to_si g n idx lit ++
    si_to_next_si g n idx ++
    si_to_next_xi g idx lit) l),
  (g.nfresh (n - 1)).2⟩

def xi_to_si (g) (n i : nat) (lit) : cnf V :=
  if i < n - 1 then
    [[lit.flip, Pos (g.nth i)]] else []
```

The function `map_with_index` applies a function to each element in a list along with its index in the list. We use `map_with_index` to access the corresponding signal variable for each literal in `l`. The function `join` flattens a list of lists into a single list.

In the recursive implementation `sc_rec`, we generate a fresh signal variable `y` at each recursive level. We also generate a second fresh variable `z` since we need to produce a clause with two adjacent signal variables, but the `gensym` given to the recursive call is the one that was produced by a single call to `fresh`. We once again omit the trivial cases.

```
def sc_rec : enc_fn V
| [l₁, l₂]          g :=
  let ⟨y, g₁⟩ := g.fresh in
  ⟨[[l₁.flip, Pos y], [Neg y, l₂.flip]], g₁⟩
| (l₁ :: l₂ :: ls) g :=
  let ⟨y, g₁⟩ := g.fresh in
  let ⟨z, _⟩ := g₁.fresh in
  let ⟨F', g₂⟩ := sc_rec (l₂ :: ls) g₁ in
  ⟨[[l₁.flip, Pos y], [Neg y, Pos z],
      [Neg y, l₂.flip]] ++ F', g₂⟩
```

The correctness proof for `sc_rec` proceeds by induction on the input list `l`. For the forward direction, the proof strategy is similar to the one for `recursive_parity`. An assignment that extends the given τ is constructed by setting the truth value for signal variable s_1 to $\tau(x_1)$. If s_1 is set to true, then the assignment that sets all signal variables to true is proved to satisfy the formula. Otherwise, s_1 is set to false in the extended assignment, and the induction hypothesis gives back an assignment σ that satisfies the recursive sub-formula. An `aite` construction finishes the proof.

Now the reverse direction. If $\tau(x_1)$ is true, then we use a lemma stating that because τ satisfies the formula, all other x_i must be false under τ (due to all signal variables being true). Otherwise, the induction hypothesis gives that $\text{AMO}(x_2, \ldots, x_n)$ is satisfied under τ, and since $\tau(x_1)$ is false, the full constraint is satisfied.

The correctness proof for the non-recursive `sc_amo` required engineering that was not present in the proof for `sc_rec`. The additional engineering contributed to the proof being 50% larger (300 vs. 200 lines).

In the forward direction, we supply an explicit truth assignment `sc_tau` that provides the truth values for all signal variables at once. A helper function `var_idx` extracts the index i for each provided signal variable. Proving that `sc_tau` satisfies the encoded formula was straightforward but tedious.

For the reverse direction, if no x_i is true under τ, then of course the AMO constraint is satisfied. Otherwise, the proof uses lemmas showing that the signal variables propagate the truth value of x_i appropriately. Because the lemmas were stated in terms of "if x_i is true, then all later x_j are false," `distinct` was used to finish the proof.

There are two main takeaways. The first is that the recursive implementation and its correctness proof are more compact than the non-recursive version's. Because the fresh variables generated at each recursive level are largely independent, the induction hypothesis can be leveraged effectively.

The second takeaway is that much of the proof overhead in the non-recursive case came from managing hypotheses about set membership among the fresh variables (i.e., that they are disjoint from the literals in `l` and the updated stock, and that they were distinct from each other). As we developed our library, we added more lemmas to ease these proof burdens, but ultimately, proving facts like `g.nth i` $\notin$ `l.take j` when `g.stock` and `l` are disjoint will persist. Future work could address this burden by automating the proving of these facts.

We implemented the AMK encoding in a non-recursive manner analogous to `sc_amo`. Since the implementation and correctness proof are so similar, we omit the details.

VI. Encoding Sudoku

To demonstrate the use of our proof library, we implemented an encoding for the Sudoku problem using AMO sub-encodings. Because the encoding was formed by composing well-behaved, correct sub-encodings, its correctness proof was only 15 lines. In addition, it uses an abstract sub-encoding `amo_enc` that can be defined to be any correct sub-encoding, which happily agrees with the mathematical view of encodings.

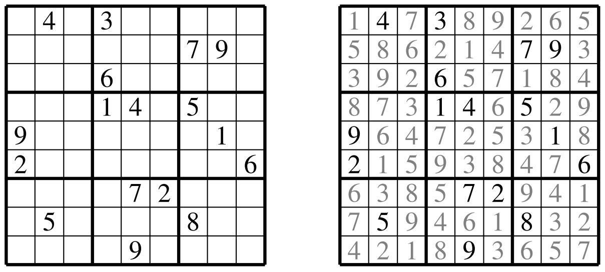
Fig. 3. A 9-by-9 Sudoku puzzle (left) and its unique solution (right). Every row, column, and 3-by-3 subgrid that compose the grid must have exactly one each of the numbers 1 through 9.

Sudoku is a classic Japanese puzzle where one must fill in a number between 1 and 9 in every cell in a 9-by-9 grid such that every row, column, and 3-by-3 subgrid comprising the grid must contain each number 1 through 9 exactly once. Numbers present in the grid from the start define a single unique solution to the puzzle. Figure 3 depicts a difficult Sudoku puzzle and its solution. The general Sudoku problem is parameterized by n, the side length for a single subgrid. In the figure, $n = 3$.

One encoding for Sudoku is to use AMO constraints for each cell, row, column, and square, along with an ALO constraint on each cell. Let $X = \{x_{r,c,k}\}$ be the set of boolean variables used in the encoding. Setting $x_{r,c,k}$ to true means placing number k in the cell in row r and column c. Each of r, c, and k run from 1 to n^2, making a total of n^6 variables. For example, the AMO constraint on the rows would look like

$$\bigwedge_{r=1}^{n^2} \bigwedge_{k=1}^{n^2} \text{AMO}(x_{r,1,k}, \ldots, x_{r,n^2,k}).$$

Implementing the Sudoku encoding in Lean using the machinery we've discussed so far seems challenging. At first blush, it looks like the `append` operation could combine all the sub-encodings together, but `append` combines encodings that share an input list of literals. In the Sudoku encoding discussed above, each ALO and AMO constraint is expressed on a *different subset* of the n^6 variables in X. So `append` won't work without some modification.

Our solution is to compose each ALO and AMO encoding function with a function `filter_by_idx` that filters out indexes in a list that don't satisfy a given predicate. This way, all of the encoding functions can be folded together, since each will extract the literals it needs. An example of one predicate we use, `is_cell_lit`, returns whether a list index corresponds to one of the literals associated with a particular cell. The filter functions help define the sub-constraints on each cell, row, column, and square on the Sudoku board.

```
def is_cell_lit (n row col : nat) := λ idx,
  idx ∈ (range (n^2)).map (λ num,
    (row * n^4) + (col * n^2) + num)
```

The function `range k` returns a list $[0, 1, \ldots, k-1]$.

(The presentation of `is_cell_lit` above elides many bookkeeping details. For instance, we use Lean's `fin` type instead of `nat`. However, the logical core is the same.)

In the spirit of Wadler's "Theorems for Free!" [44], the correctness and well-behavedness of encoding functions are preserved under operations on arbitrary lists, such as permutation and element copying and deletion, since our notion of correctness refers only to the values in the list and not the list itself. We use the theorem that `filter_by_idx`, which returns a sublist of its input, preserves an encoding function's correctness to prove the Sudoku encoding correct.

We can now define the Sudoku constraint. We note that while the constraint isn't the most efficient one possible, it is the most natural, and the redundant clauses in the encoded formula help SAT solvers in practice. The encoding is defined almost identically by swapping in the abstract encoding function `amo_enc` in place of the sub-constraints.

```
def is_valid_sudoku (n : nat) :=
let L := cart_prod (n^2) (n^2) in
fold (L.map (λ ⟨r, c⟩, is_cell_valid r c)) ++
fold (L.map (λ ⟨c, k⟩, is_row_valid c k)) ++
fold (L.map (λ ⟨r, k⟩, is_col_valid r k)) ++
fold ((cart_prod n n).zip
  (fin.range (n^2))).map
  (λ ⟨⟨sr, sc⟩, k⟩, is_subgrid_valid sr sc k)
```

We omit a check by the function `len_check`, which ensures that the constraint only accepts lists of appropriate length (of length n^6). The function `cart_prod` returns a list of pairs representing the Cartesian product of the universe of fintypes. For example, `cart_prod 2 3` $= [(0,0),(0,1),(0,2),(1,0),(1,1),(1,2)]$.

To demonstrate Lean's ability to produce these encodings, we had Lean generate and save the Sudoku encoding to a file in DIMACS format, which can then be sent to conventional SAT solvers. The file can be found in our library.

VII. Related work

We are not the first to verify SAT encodings using a proof assistant. Our work is most similar to Giljegård and Wennerbreck's library for verified SAT encodings [45], written in CakeML [46]. In their library, they verified the correctness of the naive AMO encoding, several encodings of pseudo-boolean constraints, and the Tseitin transformation for turning arbitrary propositional logic formulas into CNF. They also provide a way to translate mathematical objects (e.g., unordered sets and natural numbers) into SAT analogues, which helps with the writing of encodings. They then applied their verified encodings to logic puzzles like the n-queens problem and Sudoku variants.

Our work improves on Giljegård and Wennerbreck's library in two main regards. The first is our richer set of operations and theorems on CNF objects. For example, constructing composite truth assignments with `aite` is crucial for proving the correctness of encodings that introduce fresh variables.

The second improvement is our library's management of fresh variables. While Giljegård and Wennerbreck's library also generates fresh variables to implement the Tseitin transformation, their reasoning about fresh variables is specialized to the Tseitin transformation, as it is the only place in their library where fresh variables are used. What's more, they state that they did not implement more-efficient encodings for pseudo-boolean constraints due to the challenges of fresh variable management. Our library solves this problem.

Our work also shares similarities with Luís Cruz-Filipe, et al.'s work on an end-to-end verification of the encoding of the Pythagorean triples problem [47]. They verified the encoding, and additional symmetry breaking techniques, in Coq [48]. Their types for literals, clauses, and CNF formulas are identical to ours, and their notion of encoding correctness agrees with our definition. However, because the encoding they verified did not introduce fresh variables, they did not develop any infrastructure for managing fresh variables.

Other verification efforts are domain-specific and mainly translate other logical systems into SAT [49–51]. For example, Ishii and Fujii verify an encoding of SAT-based model checking in Coq. They formalize methods such as k-induction and property-directed reachability to check the safety of state-transition systems, and then they prove the soundness of converting safety properties expressed using these methods into SAT. By taking in a fixed number of transitions k, they express the safety properties in terms of a finite logical formula, which is then converted into propositional logic.

Our work has already seen application beyond this paper. Holliday, Norman, and Pacuit [52] used our library to verify their SAT encoding of problems in voting theory.

VIII. Conclusion and future work

Our library has laid the groundwork for formally verifying SAT encodings. The encodings we verify are efficient in practice, even at large scales, and we have made efforts to develop a framework that is general, extensible, and easy to use in practice. So far, we have verified the correctness of encodings for the parity, at-most-one, and at-most-k constraints and an encoding of Sudoku. To do so, we developed methods of generating fresh variables, constructing extended assignments, and combining sub-encodings, and we proved lemmas that allow us to reason about these operations.

Despite our progress, there is still much left to do. We plan to upgrade our library to Lean version 4, which offers better automation and linking to SAT and satisfiability modulo theory solvers. We will also rewrite our `gensym` in terms of a state monad to simplify writing encodings. Finally, many encodings are still unverified. For example, a wide number of SAT-solving applications require efficient encodings of cardinality constraints [40, 53], pseudo-boolean constraints [54, 55], and symmetry-breaking predicates [56, 57].

In the long run, our goal is to provide tools so that any claim established with a SAT solver can be fully verified from start to finish. Increasingly-complex mathematical theorems and claims about hardware and software are being reduced to propositional search problems, and these reductions are becoming more subtle and involved. Interactive theorem proving therefore has an important role to play, and our goal is to develop a library that can adequately support the task.

References

[1] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi, "Benefits of bounded model checking at an industrial setting," in *CAV*, pp. 436–453, Springer, 2001.

[2] F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, and P. Ashar, "Efficient SAT-based bounded model checking for software verification," *Theoretical Computer Science*, vol. 404, no. 3, pp. 256–274, 2008.

[3] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings* (G. Gopalakrishnan and S. Qadeer, eds.), vol. 6806 of *Lecture Notes in Computer Science*, pp. 171–177, Springer, 2011.

[4] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems* (C. R. Ramakrishnan and J. Rehof, eds.), (Berlin, Heidelberg), pp. 337–340, Springer Berlin Heidelberg, 2008.

[5] J. Brakensiek, M. J. H. Heule, J. Mackey, and D. Narváez, "The Resolution of Keller's Conjecture," in *Automated Reasoning* (N. Peltier and V. Sofronie-Stokkermans, eds.), (Cham), pp. 48–65, Springer International Publishing, 2020.

[6] M. J. H. Heule, O. Kullmann, and V. W. Marek, "Solving and verifying the Boolean Pythagorean triples problem via cube-and-conquer," in *Theory and Applications of Satisfiability Testing – SAT 2016* (N. Creignou and D. Le Berre, eds.), (Cham), pp. 228–245, Springer International Publishing, 2016.

[7] A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. Amsterdam, The Netherlands: IOS Press, 2009.

[8] S. A. Cook, "The complexity of theorem-proving procedures," in *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, (New York, NY, USA), pp. 151–158, ACM, 1971.

[9] R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell, "Efficient generation of unsatisfiability proofs and cores in SAT," in *LPAR*, pp. 16–30, 2008.

[10] A. Van Gelder, "Producing and verifying extremely large propositional refutations - have your cake and eat it too," *Ann. Math. Artif. Intell.*, vol. 65, no. 4, pp. 329–372, 2012.

[11] M. J. H. Heule, W. A. Hunt, Jr., and N. Wetzler, "Verifying refutations with extended resolution," in *International Conference on Automated Deduction (CADE)*, vol. 7898 of *LNAI*, pp. 345–359, Springer, 2013.

[12] E. I. Goldberg and Y. Novikov, "Verification of proofs of unsatisfiability for CNF formulas," in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pp. 10886–10891, IEEE, 2003.

[13] M. J. H. Heule, W. A. Hunt, Jr., and N. Wetzler, "Trimming while checking clausal proofs," in *Formal Methods in Computer-Aided Design (FMCAD)*, pp. 181–188, IEEE, 2013.

[14] L. Cruz-Filipe, M. J. H. Heule, W. A. Hunt, M. Kaufmann, and P. Schneider-Kamp, "Efficient certified RAT verification," in *Automated Deduction – CADE 26* (L. de Moura, ed.), (Cham), pp. 220–236, Springer International Publishing, 2017.

[15] L. Cruz-Filipe, J. Marques-Silva, and P. Schneider-Kamp, "Efficient certified resolution proof checking," in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 10205, pp. 118–135, 2017.

[16] M. J. H. Heule, W. Hunt, M. Kaufmann, and N. Wetzler, "Efficient, verified checking of propositional proofs," in *Interactive Theorem Proving* (M. Ayala-Rincón and C. A. Muñoz, eds.), (Cham), pp. 269–284, Springer International Publishing, 2017.

[17] Y. K. Tan, M. J. H. Heule, and M. O. Myreen, "cake_lpr: Verified propagation redundancy checking in CakeML," in *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II* (J. F. Groote and K. G. Larsen, eds.), vol. 12652 of *Lecture Notes in Computer Science*, pp. 223–241, Springer, 2021.

[18] L. M. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer, "The Lean theorem prover (system description)," in *Conference on Automated Deduction (CADE) 2015* (A. P. Felty and A. Middeldorp, eds.), pp. 378–388, Springer, Berlin, 2015.

[19] A. Leventi-Peetz, O. Zendel, W. Lennartz, and K. Weber, "CryptoMiniSat switches-optimization for solving cryptographic instances," in *Proceedings of Pragmatics of SAT 2015 and 2018* (D. L. Berre and M. Järvisalo, eds.), vol. 59 of *EPiC Series in Computing*, pp. 79–93, EasyChair, 2019.

[20] M. Soos, K. Nohl, and C. Castelluccia, "Extending SAT solvers to cryptographic problems," in *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings* (O. Kullmann, ed.), vol. 5584 of *Lecture Notes in Computer Science*, pp. 244–257, Springer, 2009.

[21] A. Graça, I. Lynce, J. Marques-Silva, and A. L. Oliveira, "Efficient and accurate haplotype inference by combining parsimony and pedigree information," in *Algebraic and Numeric Biology* (K. Horimoto, M. Nakatsui, and N. Popov, eds.), pp. 38–56, Springer Berlin Heidelberg, 2012.

[22] M. Soos and K. S. Meel, "BIRD: engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting," in *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference,*

IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019, pp. 1592–1599, AAAI Press, 2019.

[23] J. P. Marques-Silva and I. Lynce, "Towards robust CNF encodings of cardinality constraints," in *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings* (C. Bessiere, ed.), vol. 4741 of *Lecture Notes in Computer Science*, pp. 483–497, Springer, 2007.

[24] N. Manthey, M. J. H. Heule, and A. Biere, "Automated reencoding of Boolean formulas," in *Hardware and Software: Verification and Testing* (A. Biere, A. Nahir, and T. Vos, eds.), (Berlin, Heidelberg), pp. 102–117, Springer Berlin Heidelberg, 2013.

[25] M. J. H. Heule, M. Kauers, and M. Seidl, "New ways to multiply 3×3-matrices," *J. Symb. Comput.*, vol. 104, pp. 899–916, 2021.

[26] W. Nawrocki, Z. Liu, A. Fröhlich, M. J. H. Heule, and A. Biere, "XOR local search for Boolean brent equations," in *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings* (C. Li and F. Manyà, eds.), vol. 12831 of *Lecture Notes in Computer Science*, pp. 417–435, Springer, 2021.

[27] S. A. Weaver, H. J. Roberts, and M. J. Smith, "XOR-satisfiability set membership filters," in *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings* (O. Beyersdorff and C. M. Wintersteiger, eds.), vol. 10929 of *Lecture Notes in Computer Science*, pp. 401–418, Springer, 2018.

[28] G. V. Bard, N. T. Courtois, and C. Jefferson., "Efficient methods for conversion and solution of sparse systems of low-degree multivariate polynomials over GF(2) via SAT-solvers." Cryptology ePrint Archive, Report 2007/024, 2007.

[29] M. Gwynne and O. Kullmann, "A framework for good SAT translations, with applications to CNF representations of XOR constraints," 2014.

[30] M. Gwynne and O. Kullmann, "On SAT representations of XOR constraints," in *Language and Automata Theory and Applications - 8th International Conference, LATA 2014, Madrid, Spain, March 10-14, 2014. Proceedings* (A. Dediu, C. Martín-Vide, J. L. Sierra-Rodríguez, and B. Truthe, eds.), vol. 8370 of *Lecture Notes in Computer Science*, pp. 409–420, Springer, 2014.

[31] G. S. Tseitin, "On the complexity of derivation in propositional calculus," in *Automation of Reasoning 2* (J. Siekmann and G. Wrightson, eds.), pp. 466–483, Springer, 1983.

[32] W. Küchlin and C. Sinz, "Proving consistency assertions for automotive product data management," *Journal of Automated Reasoning*, vol. 24, no. 1, pp. 145–163, 2000.

[33] B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J. P. Warners, "Radio link frequency assignment," *Constraints*, vol. 4, no. 1, pp. 79–89, 1999.

[34] M. Jose and R. Majumdar, "Cause clue clauses: Error localization using maximum satisfiability," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, p. 437–446, Association for Computing Machinery, 2011.

[35] R. Asín Achá and R. Nieuwenhuis, "Curriculum-based course timetabling with SAT and MaxSAT," *Annals of Operations Research*, vol. 218, no. 1, pp. 71–91, 2014.

[36] O. Bailleux and Y. Boufkhad, "Efficient CNF encoding of Boolean cardinality constraints," in *Principles and Practice of Constraint Programming – CP 2003* (F. Rossi, ed.), (Berlin, Heidelberg), pp. 108–122, Springer Berlin Heidelberg, 2003.

[37] A. Frisch, T. Peugniez, A. Doggett, and P. Nightingale, "Solving non-Boolean satisfiability problems with stochastic local search: A comparison of encodings," *Journal of Automated Reasoning*, vol. 35, pp. 143–179, 01 2005.

[38] W. Klieber and G. Kwon, "Efficient CNF encoding for selecting 1 from N objects," in *Fourth Workshop on Constraint in Formal Verification (CFV)*, 2007.

[39] V.-H. Nguyen and S. T. Mai, "A new method to encode the at-most-one constraint into SAT," in *Proceedings of the Sixth International Symposium on Information and Communication Technology*, SoICT 2015, (New York, NY, USA), p. 46–53, Association for Computing Machinery, 2015.

[40] C. Sinz, "Towards an optimal CNF encoding of Boolean cardinality constraints," in *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings* (P. van Beek, ed.), vol. 3709 of *LNCS*, pp. 827–831, Springer, 2005.

[41] The mathlib community, "The Lean mathematical library," in *Certified Programs and Proofs (CPP) 2020* (J. Blanchette and C. Hritcu, eds.), pp. 367–381, ACM, 2020.

[42] N. de Bruijn, "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem," *Indagationes Mathematicae (Proceedings)*, vol. 75, no. 5, pp. 381–392, 1972.

[43] A. Filinski, "Normalization by evaluation for the computational lambda-calculus," in *Proceedings of the 5th International Conference on Typed Lambda Calculi and Applications*, TLCA'01, (Berlin, Heidelberg), pp. 151––165, Springer-Verlag, 2001.

[44] P. Wadler, "Theorems for free!," in *Conference on Functional Programming Languages and Computer Architecture*, 1989.

[45] S. Giljegård and J. Wennerbreck, "Puzzle solving with proof," Master's thesis, Chalmers University of Technology, University of Gothenburg, 2021.

[46] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens,

"CakeML: A verified implementation of ML," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, (New York, NY, USA), pp. 179–191, Association for Computing Machinery, 2014.

[47] L. Cruz-Filipe, J. Marques-Silva, and P. Schneider-Kamp, "Formally verifying the solution to the Boolean Pythagorean triples problem," *J. Autom. Reason.*, vol. 63, no. 3, pp. 695–722, 2019.

[48] Y. Bertot and P. Castéran, *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[49] K. Anuarul Hoque, O. A. Mohamed, S. Abed, and M. Boukadoum, "An automated SAT encoding-verification approach for efficient model checking," in *2010 International Conference on Microelectronics*, pp. 419–422, 2010.

[50] D. Ishii and S. Fujii, "Formalizing the soundness of the encoding methods of SAT-based model checking," in *International Symposium on Theoretical Aspects of Software Engineering, TASE 2020, Hangzhou, China, December 11-13, 2020* (T. Aoki and Q. Li, eds.), pp. 105–112, IEEE, 2020.

[51] M. Abdulaziz and F. Kurz, "Formally verified SAT-based AI planning," 2020.

[52] W. H. Holliday, C. Norman, and E. Pacuit, "Voting theory in the Lean theorem prover," in *Logic, Rationality, and Interaction: 8th International Workshop, Lori 2021, Xi?an, China, October 16?18, 2021, Proceedings* (S. Ghosh and T. Icard, eds.), pp. 111–127, Springer Verlag, 2021.

[53] J.-C. Régin, "Generalized arc consistency for global cardinality constraint," in *14th National Conference on Artificial Intelligence (AAAI 1996)*, vol. 1, pp. 209–215, AAAI Press / The MIT Press, 1996.

[54] O. Bailleux, Y. Boufkhad, and O. Roussel, "A translation of pseudo-Boolean constraints to SAT," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, pp. 183–192, 2006. Special Issue on SAT 2005 competition and evaluations.

[55] N. Eén and N. Sörensson, "Translating Pseudo-Boolean Constraints into SAT," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, pp. 1–25, 2006. Special Issue on SAT 2005 competition and evaluations.

[56] F. A. Aloul, K. A. Sakallah, and I. L. Markov, "Efficient symmetry breaking for Boolean satisfiability," *IEEE Trans. Computers*, vol. 55, no. 5, pp. 549–558, 2006.

[57] J. Crawford, M. Ginsberg, E. Luks, and A. Roy, "Symmetry-breaking predicates for search problems," in *Proc. KR'96, 5th Int. Conf. on Knowledge Representation and Reasoning*, pp. 148–159, Morgan Kaufmann, 1996.