Verifying Sat Encodings in Lean

Verifying Sat Encodings in Lean

Cayden R. Codel

May 6, 2022

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

Marijn Heule, advisor, chair
Jeremy Avigad, advisor
Bryan Parno

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Authored and distributed by: Cayden R. Codel
E-mail: ccodel@andrew.cmu.edu
Website: www.crcodel.com

Keywords: Formal verification, Lean, theorem proving, Sat encodings

This research has been supported by the Hoskinson Center for Formal Mathematics at Carnegie Mellon University.

Abstract

Satisfiability (SAT) solvers are versatile tools that can help solve a wide array of problems. For satisfiable instances, solvers output candidate solutions that can be efficiently checked. For unsatisfiable instances, many solvers emit proofs of unsatisfiability that can also be efficiently and formally checked. Thus, we can formally check SAT solver results for any given problem instance.

However, many applications have problem instances that are *encoded* into SAT, rather than expressed in SAT natively. For any problem instance, there are often many choices of encoding. These encodings range in size and complexity, and the choice of encoding has a direct impact on solver performance. But crucially, if the encoding is not correct, then solver results are not trustworthy. Formal correctness proofs of the encodings ensure that trust.

In this work, we introduce a library for formally verifying SAT encodings. We present formal verifications of encodings for the n-variable XOR function and the at-most-one function. We also verify an encoding of the at-most-k function. We completed our proofs using the interactive theorem prover Lean. The methods we developed of formally verifying these encodings extend beyond their applications, and so this library serves as a basis for future encoding verification efforts.

Acknowledgements

My work would not be possible without my advisors Profs. Heule and Avigad. It is a rare opportunity to be co-advised by an expert in SAT solving and by a developer of the Lean theorem prover. Whenever I had a question, inevitably one of them would have an answer. I hope to continue collaborations with them and their students as I study at CMU as a PhD student. Thank you for your patience, your expertise, and your mentorship.

I also thank the many professors with whom I interacted as I explored my research interests. Profs. Salakhutdinov, Harpstead, and Koedinger were generous with their time as I sampled a project or a slice of their work. While my research home may not be machine learning or human-computer interaction, without joining them and their students, I would not have found SAT solving and theorem proving.

Learning how to use Lean was difficult. I appreciate the knowledge and spirit of giving on the Lean Zulip community. Without their discussions, I would have struggled to make my proofs compile much more than I did.

Finally, I thank my friends and family for their support and interest in my work. Part of the joy of research is sharing it with others. Thank you for having a sparkle in your eye as I showed you mine.

Contents

The greatest mistake is to imagine that we never err.

Thomas Carlyle

1 Introduction

One day, you find a golden lamp in your dusty attic, but you don't recall ever owning such a lamp. Scenes from Disney's *Aladdin* flash in your mind as you pick it up and give it a rub. At your touch, the lamp shakes in your hand and spews forth multicolored fog and smoke. A booming voice announces that you awoke the oracle of the lamp and that it will answer any question you ask it. But beware! You must be sure you have asked the right question, for the oracle will answer only once. You consider your options. You have many questions you would like to ask the oracle, but how can you ask the right question? In asking for the meaning of life, how can you ensure the oracle won't give a rote dictionary definition and disappear in a puff of smoke? The problem is difficult, but you feel that if you express your thoughts precisely, you can get the answers you desire.

While the attic scene is fantastical, computer scientists face similar situations in a wide range of research problems. Often, researchers and engineers phrase difficult problems in terms of another and use a *black-box algorithm*—an oracle—to find an answer. Part of the challenge of using an oracle lies in translating, or *encoding*, the problem efficiently and correctly. If the encoded version becomes too large, then a computationally-bound oracle may take too long to answer. Or worse, if the encoded version doesn't accurately represent the original problem, then the oracle's answers are of no help.

Satisfiability (Sat) solvers are powerful and versatile tools that are commonly used as a black-box. They solve instances of the Sat problem, which asks whether there exists a way to assign true and false values to Boolean variables to satisfy a formula. Sat is NP-complete [12], and thus many computationally challenging problems can be reduced to a Sat instance and given to a solver. As just a sampling of their prevalence in computer science research, solvers are crucial in termination analysis of term-rewriting systems [25, 59], for verifying hardware and software [13, 37], as internal tools for SMT solvers [6, 20], for the bounded-model checking of `C` programs [11], and in resolving longstanding open problems in mathematics [10, 36]. As Donald Knuth put it, Sat solving is a "killer app."

One benefit of using a Sat solver as an oracle is the ability to verify results. If a solver reports a satisfying assignment, then the candidate solution can be efficiently checked against the input formula. Conversely, many solvers emit a certificate of unsatisfiability when they report that there is no satisfying assignment. These certificates are written in a formal proof system, such as resolution, that give a step-by-step proof of unsatisfiability [3, 34, 57]. A proof checker can then machine-verify that the certificate is correct [28, 33]. Proof checkers are often simple pieces of software, and many checkers have been formally verified [16, 17, 32, 54].

The end result of the ability to verify solver results is that the SAT toolchain is *trustworthy*. Even if the solver has bugs, any result produced by the solver can be machine-checked in an efficient way. The offline checking of solver results allows the solver to focus on algorithmic and data structure optimization. However, solvers incur significant overhead when producing unsatisfiability certificates [56]. The alternative is to formally verify the solver itself, so any emitted result is automatically trustworthy. A few solvers have been verified [41, 47].

Yet even if the SAT toolchain is trustworthy, solvers that are given unverified encodings still produce untrustworthy results. Just as how asking the oracle in the attic a poorly phrased question can give back a nonsense answer, so too can an error in translation give back incorrect SAT results. One way to increase confidence in solver results is to provide a proof of correctness of the encoding used. Pen-and-paper proofs suffice in a pinch, but formal proofs written in a proof assistant are machine-verifiable and thus more trustworthy.

Researchers have already begun to use proof assistants to verify SAT encodings. Cruz-Filipe, Marques-Silva, and Schneider-Kamp used Coq [7] to verify the base encoding, the symmetry breaking used to simplify the encoding, and the proof checker used in resolving the Pythagorean triples problem [18]. Hoque et al. used Isabelle/HOL [46] to verify the encoding of a multiway decision graph into SAT [2]; the graph is used for bounded model checking applications. Ishii and Fujii used Coq to verify a SAT-based IC3 algorithm, and Abdulaziz and Kurz verified SAT-based planning for AI in HOL [1]. Giljegård and Wennerbreck used HOL4 [50] to verify a set of CNF encoding functions [27].

In this thesis, we add to the line of work in verifying SAT encodings by introducing our own proof library, written using the proof assistant Lean [21]. Our library formally verifies encodings of the Boolean XOR function, the at-most-one function, and the at-most-k function. In addition to these results, our library contains supporting lemmas and data structures that will assist in future verification efforts.

The thesis is organized as follows. Chapter 2 defines what it means for a formula to encode a function. Chapter 3 discusses the Lean theorem prover and introduces Lean's syntax. Chapter 4 introduces our proof library. We introduce and prove correct the XOR function in Chapter 5. We do the same for the at-most-one and at-most-k functions in Chapter 6. We conclude in Chapter 7.

The difference between the right word and the almost right word is really a large matter — it's the difference between lightning and a lightning bug.

Mark Twain

2 SAT encodings

So you've decided to use a SAT solver to solve your problem. Great choice! Your problem needs to be encoded into SAT first. But what does it mean to encode your problem?

In this chapter, we define the notational machinery we use to talk about Boolean formulas, satisfiability, and SAT encoding, and we discuss how to encode a problem into SAT.

2.1 Preliminaries

SAT solvers operate on formulas in propositional logic known as SAT instances. These formulas are comprised of Boolean literals joined by logical connectives. A *Boolean variable* is a variable that ranges 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. If $x = \top$, then $\overline{x} = \bot$, and vice versa. Common logical connectives are AND ($\wedge$), OR ($\vee$), implication ($\rightarrow$), and logical equivalence ($\leftrightarrow$), with the usual truth tables for each. Solvers attempt to answer the question of whether there exists a *truth assignment* to the Boolean variables that satisfies the formula. Such a truth assignment is called a *satisfying assignment*. When F admits a satisfying assignment, we call F *satisfiable*. Many solvers output a satisfying assignment when they determine that a formula F is satisfiable.

As an example, consider the following formula:

$$F = (x_1 \vee x_2) \wedge (\overline{x}_1 \vee x_3) \wedge (\overline{x}_2 \vee \overline{x}_3). \tag{2.1}$$

F is satisfiable, as $\tau = \{x_1 = \top, x_2 = \bot, x_3 = \top\}$ is a satisfying assignment. We write $\tau(F) = \top$ when τ satisfies F, and $\tau(F) = \bot$ if not.

Many SAT solvers only accept formulas in *conjunctive normal form* (CNF) as valid input. A formula is in CNF if it is a conjunction of *clauses*, each of which is a disjunction of Boolean literals. Formula 2.1 is an example of a CNF formula.

While the requirement that input formulas be in CNF may seem restrictive, there is a well-known result that any formula written with binary connectives can be transformed to a logically equivalent one in CNF. However, the transformation may result in an exponential blowup in the number of clauses, which impedes solver performance. As a result, researchers have found ways of introducing fresh auxiliary variables to transform formulas into CNF while keeping formula size linear in the input size.

Let $\text{vars}(\cdot)$ give the set of variables in a formula. For Formula 2.1, $\text{vars}(F) = \{x_1, x_2, x_3\}$. We overload $\text{vars}(\cdot)$ for truth assignments τ to mean the set of variables that τ is defined over.

If $\text{vars}(\tau) \subsetneq \text{vars}(F)$, then we call τ a *partial assignment* for F. When $\text{vars}(\tau) \supseteq \text{vars}(F)$, then τ is a *full assignment* for F. If σ is a truth assignment such that $\text{vars}(\tau) \subseteq \text{vars}(\sigma)$ and $\tau(x) = \sigma(x)$ whenever $x \in \text{vars}(\tau)$, then we say that σ *extends* τ.

2.2 An example encoding

A great many problems are not written in terms of Sat instances. Yet because Sat is an NP-complete problem [12], it is common to reduce these (possibly NP-complete) problems to Sat. These reductions are accomplished via an *encoding*, which is an explicit algorithmic transformation of the objects and constraints in the problem into a Boolean formula. A *correct* encoding allows for the construction of a solution to the original problem when given a satisfying assignment to the Sat instance.

To demonstrate how an NP-complete problem can be reduced to Sat, we encode the problem of finding a Hamiltonian cycle in a graph into Sat.

Let $G = (V, E)$ be a simple undirected graph, where V is the set of vertices and $E \subseteq V \times V$ is the set of edges. Let $n := |V|$. A Hamiltonian cycle is a single cycle through G that visits every vertex exactly once. Formally, a Hamiltonian cycle is a sequence of vertices

$$S = v_1 v_2 \ldots v_{n-1} v_n$$

where S is a permutation of V and for all $1 \leq i \leq n$, if v_i, v_{i+1} are two adjacent vertices in S, then $(v_i, v_{i+1}) \in E$ (where the i are mod n). The Hamiltonian cycle (hc) problem asks whether there exists a sequence S that satisfies these properties and, in the search version, requires one to produce such an S, if one exists. The hc problem is well-known to be NP-complete.

We start our encoding with the variables of our formula. We define a time-indexed set of Boolean variables for each vertex:

$$\{x_{v,1}, x_{v,2}, \ldots, x_{v,n}\} \qquad \forall v \in V.$$

We interpret $x_{v,t} = \top$ to mean that vertex v is the tth visited vertex in our cycle. We now translate the constraints of the Hamiltonian cycle into Sat on these variables.

First, we require that for any vertex v visited in our cycle, there is some neighbor u that we visited in the preceding time step (or at the nth time step, for the first vertex). Thus, for all $v \in V$ and for all $1 \leq t \leq n$,

$$x_{v,t} \rightarrow (x_{u_1,t-1} \vee x_{u_2,t-1} \vee \cdots \vee x_{u_k,t-1}) \equiv \left(\overline{x}_{v,t} \vee \bigvee_{i=1}^{k} x_{u_i,t-1} \right), \tag*{$\lfloor$2.2}$$

where the u_i are the k neighbors of v arranged in some order and where $t-1$ is positive mod n.

We also need to enforce that each vertex is visited exactly once in any Hamiltonian cycle with some kind of encoding. Let's call this encoding ExactlyOne. So for all $v \in V$, we require that

$$\textsc{ExactlyOne}(x_{v,1}, \ldots, x_{v,n}) \tag*{$\lfloor$2.3}$$

be satisfied. As we will see in Chapter 6, there are several ways in which ExactlyOne encodings can be defined.

Put those two sets of constraints together, and we have a formula encoding the hc problem. If each of constraints 2.2 and 2.3 are written in CNF, then we have a CNF formula ready to give to a Sat solver.

Assuming a Hamiltonian cycle S exists, the solver should report a satisfying assignment on the $x_{v,t}$. We can construct a cycle by setting v_i to be the $v \in V$ with $x_{v,i} = \top$, for each $1 \leq i \leq n$. Thus, we have a way to construct a solution from a satisfying assignment to the encoded instance.

The above encoding of the hc problem was a straightforward one, but not the only possible one. In fact, if ExactlyOne produces $O(n^2)$ clauses for an n-variable input, then our encoding requires $O(n^3)$ clauses. For graphs above small n (say, $n = 100$), such a large encoding will cause solvers to time out or require too much memory. Thus, it is crucial that more compact encodings are used. Others exist based on different sets of variables, such as binary adders or linear-feedback shift registers [60]. Strong experimental results show that only by combining the best aspects of a variety of hc encodings can large graph benchmarks become tractable for state-of-the-art solvers [31]. The encoding is complicated, which makes a proof of correctness all the more valuable.

But from a mathematical point of view, the choice of encoding does not matter. All we require is that the encoding be correct—that it accurately represents the constraints of the problem, and that given a satisfying assignment to the encoded formula, we can construct a solution to the original problem. A formal proof can ensure correctness.

2.3 Encoding as a mathematical object

We have seen in the previous section that hc can be reduced to Sat via an encoding. But not all encodings are like the one from the previous section. For example, auxiliary variables not associated with objects in the problem are often introduced to reduce the number of clauses in the formula. Since encodings can take many forms, and since we wish to formally prove that encodings are correct, we need a mathematical foundation for what an encoding is.

In this work, we concern ourselves with the encodings of n-ary Boolean relations. Our assumption may seem a restriction, but as we saw in Section 2.2, by relating mathematical objects to sets of Boolean variables, we can easily transform our problem into one in terms of a Boolean relation.

Let $x_1, \ldots, x_n$ represent the inputs to an n-ary Boolean relation R, and let F be a formula. When $\text{vars}(F) \subseteq \{x_1, \ldots, x_n\}$, then we say that F encodes R if and only if it *defines* it: for every full assignment τ on $x_1, \ldots, x_n$, we require $R(\tau(x_1), \ldots, \tau(x_n))$ when $\tau(F) = \top$. Yet F may introduce additional variables. The following definition handles the more general case.

Definition 2.1 (Encoding a Boolean relation)**.** *Let R be a Boolean relation, F be a formula, and $x_1, \ldots, x_n$ be variables representing the inputs to R. Then F* encodes *R if and only if the following holds: for every assignment τ to $x_1, \ldots, x_n$, $R(\tau(x_1), \ldots, \tau(x_n))$ 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 R 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.

A special case of Definition 2.1 is where R is defined by a Boolean function f whose inputs are in the relation exactly when $f(\tau(x_1), \ldots, \tau(x_n)) = \top$.[1] However, using a relation in the definition is more general and allows us to prove encodings of relations whose defining Boolean function is the set of inputs on which it evaluates to false, for example. Our correctness proofs are all based on the former special case, so unless otherwise noted, we refer to an encoding of a Boolean function as the encoding of the relation it defines. At some points, we note some encoding tricks to achieve an encoding of the negated function as well.

Definition 2.1 gives us a way to prove that formulas encode relations, and thus to prove that encodings are correct. But once we have established that a formula F_1 defines a relation R, we may find it easier to prove that a different formula F_2 is correct by relating it to F_1 rather than to R. Perhaps F_1 and F_2 share clauses.

So suppose that F_1 and F_2 are formulas, viewed as encoding Boolean relations whose inputs are represented by a set of variables S. In general, F_1 and F_2 may contain other variables as well. The next definition says that they encode the same relation.

Definition 2.2 (S-equivalent)**.** *Let F_1 and F_2 be Boolean formulas and let S be any set of variables. Then F_1 and F_2 are S-equivalent if and only if the following holds: for every assignment τ such that* $\mathrm{vars}(\tau) = S$*, F_1 is satisfiable by an assignment extending τ if and only if F_2 is satisfiable by an assignment extending τ.*

In quantified propositional logic, the definition is equivalent to saying that $\exists y_1, \ldots, y_m\ F_1$ is logically equivalent to $\exists z_1, \ldots, z_\ell\ F_2$, where $y_1, \ldots, y_m, z_1, \ldots, z_\ell$ are the variables that are not contained in S. When S contains all the variables in F_1 and F_2, i.e. $S \supseteq \mathrm{vars}(F_1) \cup \mathrm{vars}(F_2)$, saying that F_1 and F_2 are S-equivalent is the same as saying that they are logically equivalent. At the other extreme, when $S = \varnothing$, saying that F_1 and F_2 are S-equivalent is the same as saying that F_1 and F_2 are equisatisfiable. The following is easy to check:

Lemma 2.3. *Suppose F_1 encodes an n-ary Boolean relation R, where the inputs of R are represented by $x_1, \ldots, x_n$. If F_1 and F_2 are $\{x_1, \ldots, x_n\}$-equivalent, then F_2 encodes R also.*

2.4 Encodings in practice

Any problem we wish to encode into SAT may admit many encodings. If each encoding is correct, then a true oracle does not care which we choose. But in practice, solvers are sensitive to the choice of encoding. Most state-of-the-art solvers only accept formulas in CNF, and only in DIMACS format, which is a standard input format where literals are represented as signed integers.[2] Solvers also tend to perform better (and time out less) on compact and arc-consistent encodings, which we define below. Since solvers are computationally-bound objects, researchers prefer these encodings.

[1] In fact, for any relation R, there is a defining Boolean function f with this property.

[2] The DIMACS format is defined at https://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index-seo.php/SATLINK____DIMACS.

Compact encodings

Two metrics used to determine the utility of various encodings are the number of variables and the number of clauses in formulas that encode an n-variable Boolean relation. For example, our encoding from Section 2.2 required n variables and $O(n^3)$ clauses. By introducing auxiliary variables, one can reduce the number of clauses in the encoding, at the cost of increasing the number of variables.

Researchers have engaged in the empirical study of whether encodings with fewer clauses and variables are truly more effective. Marques-Silva and Lynce [48] showed that making encodings of cardinality constraints more compact resulted in a clear performance boost. Nguyen et al. showed a solver preference for smaller encodings for the at-most-one function [45]. Interviews conducted by Björk indicate that SAT experts prefer compact encodings for their domain-specific problems, although the search for an optimal encoding for any fixed class of problems or solver is not straightforward [8]. The general sentiment is that while the optimal encoding for any single instance is problem-dependent, encodings with fewer clauses tend to outperform direct encodings with a super-linear number of clauses.

As a result, several methods for reducing formula size have been investigated. Bounded variable elimination [22] is one such crucial processing technique that shrinks the size of the encoding to boost solver performance. Another is the Tseitin transformation [55], which transforms formulas into equisatisfiable ones via the introduction of auxiliary variables. The transformed formulas have length linear in the input formula size. We will encounter the Tseitin transformation in Chaper 5.

Consistency and arc-consistency

Many SAT solvers use internal heuristics to determine whether a candidate partial assignment is valid for an input formula. One common heuristic is unit propagation, which continually extends the partial assignment based on any present unit clauses until fixpoint. Some encodings behave well under unit propagation and so are better choices for solvers that use unit propagation. Such encodings are called *consistent* and *arc-consistent*.

First, let us discuss how unit propagation works. Assume that our input formula F is in CNF. A (disjunctive) clause $C \in F$ is unit under a partial assignment τ if one literal x is unassigned by τ while all other literals are assigned to false. Because C is a disjunction, the only way to satisfy C is to make x true. We can thus extend τ by setting $x = \top$.

Once τ has been extended, we can remove any clause in F that contains x, as those clauses are now satisfied. Instances of $\overline{x}$ are removed, as now $\overline{x} = \bot$ and so cannot satisfy any clauses it appears in. Clauses that become unit induce another round of unit propagation; clauses that become empty signal that a contradiction has been reached, and that the partial assignment cannot be correct. Unit propagation proceeds until a contradiction is reached or until all remaining clauses are not unit.

An encoding of a Boolean function is *consistent* if, for any partial input to the Boolean function that cannot be extended to an input that makes it true, unit propagation on the encoded formula using the partial input as a partial assignment should result in a conflict. For example, if $n \geq 3$ and our Boolean function f is

$$f := x_1 + x_2 + \cdots + x_n \leq 2,$$

then if our partial input sets $x_1 = x_2 = x_3 = \top$, it is clear that no full extended input will cause f to evaluate to true. A consistent encoding for f would take $\tau = \{x_1 = x_2 = x_3 = \top\}$ as a partial assignment and derive a contradiction via unit propagation.

An encoding of a Boolean function is *arc-consistent* if it is (1) consistent and (2) unit propagation on the encoding using the partial input as a partial assignment eliminates the extensions to the partial input that would make the function evaluate to false. In the above example, if a partial assignment $\tau = \{x_1 = x_2 = \top\}$ is given to an arc-consistent encoding for f, then all other x_i would be set to $\bot$ under unit propagation.

We note that while arc-consistency is an important aspect for encodings to have, as arc-consistent encodings generally achieve better solver performance than direct encodings [26], our proof library presented in Chapter 4 does not prove arc-consistency. However, the encodings that we do consider in our library happen to be arc-consistent. Proving that these encodings are, in fact, arc-consistent is a line of future work.

3 Lean Primer

Let's start at the very beginning,
a very good place to start.
Rodgers and Hammerstein, *The Sound of Music*

Our library of verified Sat encodings uses the Lean 3 interactive theorem prover [21]. In this chapter, we describe Lean's relevant features and syntax.

Readers that are already familiar with Lean can skip this chapter. Readers familiar with a different theorem prover, such as Coq [7] or Isabelle/HOL [46], may wish to skim this chapter, as Lean's syntax may differ slightly from other theorem provers'.

3.1 Theorem provers, generally

The study of formal mathematical and logical systems often concerns itself with proof— how to find one, how to express one, and how to check one. *Theorem provers* are tools that leverage the power of computers to automate aspects of proof search and proof checking.

A theorem prover reasons about and within a *formal system*, which is a set of axioms and reasoning rules which allow true statements to be derived from other true statements. Peano's axioms for natural numbers is a famous set of five axioms. An example of a reasoning rule is the law of *modus ponens* (or implication elimination), which states that if one has a proof of $A \supset B$ and a proof of A, one can produce a proof of B:

$$\frac{A \supset B \qquad A}{B}$$

Theorems are statements written in the formal system that can be proved from the axioms, the reasoning rules, and a possibly empty set of additional assumptions. Proofs of theorems can use lemmas, but each lemma must also be proved from the axioms, rules, and assumptions, and thus any theorem can be unrolled into its basic components.

Theorem provers attempt to find proofs using the axioms and rules of the formal system, and possibly also supporting lemmas and theorems available in a *proof library*. At any point, many possible axioms or rules may apply to the statement, and the theorem prover must choose which to use. Without human intervention, a theorem prover may have to try all possible options to find a proof. Theorem provers that perform proof search without human interaction are called *automated theorem provers* (atps). Vampire [39] is an example.

Theorem provers that depend on human interaction are called *interactive theorem provers* (itps) or *proof assistants*. Lean, Coq, and Isabelle/HOL are all examples of itps. Proof assistants require that a statement, or *goal*, be proven in an incremental fashion, where each step is an application of an axiom, rule, or lemma. As a result, the final proof object can be machine-checked for soundness.

Of course, that's not to say that ITPs can't automate proof search. Lean's `simp`, Coq's `simpl`, and Isabelle's `sledgehammer` tactics all mechanically search through a selected subset of theorems in an attempt to finish the proof, or at least to make forward progress. ITPs also sometimes incorporate tools such as linear arithmetic solvers or SMT solvers to assist in advancing or finishing proofs.

3.2 Lean

Lean is an ITP based on the calculus of inductive constructions [14, 15]. Specifically, its logical foundation a version of dependent type theory with inductive types, type universes, and an impredicative type of propositions. By default, Lean's core logic is constructive, but classical logic proofs can be enabled. Lean's type checker runs on a small trusted kernel written in `C++`.

There are several available versions of Lean. We use Lean 3 and its associated proof library, **`mathlib`** [42], in this thesis. At the time of publication, Lean 4 is currently in development. Both versions can be downloaded for free on the Lean website. **`mathlib`** can be found on GitHub. Unless otherwise noted, the following description of Lean applies to Lean 3, although much of it applies to Lean 4 as well.

Lean defines a small set of types that are common in functional programming and in mathematics, including natural numbers, lists, sets, and strings. Lean's elaborator is powerful enough to detect "holes" left in proofs where types can be inferred.

Figure 3.1 shows a simple tactic proof where if there are two elements in a list that aren't equal to each other, then the list has length at least two. Let us examine the proof to understand Lean's syntax and how tactics work.

```
theorem length_ge_two {α : Type*} {a b : α} (a ≠ b) {l : list α} :
  a ∈ L → b ∈ l → 2 ≤ length l :=
begin
  intros ha hb,
  cases l with x xs,
  { exact absurd ha (not_mem_nil _) },
  { cases xs with y ys,
    { rw mem_singleton at ha hb,
      rw ← hb at ha,
      contradiction },
    { rw [length, length, add_assoc, one_add_one_eq_two],
      exact le_add_self } }
end
```

Figure 3.1: A simple proof that a list with two distinct elements is at least two elements long.

Theorems in Lean begin with `theorem` or `lemma` and then an identifying name. What follows is a series of assumptions or hypotheses, and then the statement of the theorem. `length_ge_two` mixes explicit and implicit hypotheses. The hypotheses that α is a type, that `a` and `b` are elements of type α, and that `l` is a list whose elements are of type α are implicit, indicated by the curly braces {}. The hypothesis that `a ≠ b` is explicit, indicated by the parentheses (). Since `a ≠ b` uses the fact that `a` and `b` are of type α, and thus that α is a type, then the latter hypotheses can be left implicit. Lean's elaborator can infer the latter hypotheses given one of the form `a ≠ b`. The hypothesis that `l` is a list of type α is implicit because `l` is mentioned in the theorem statement. It is convention to leave assumptions implicit if they appear in later hypotheses; doing so reduces the number of arguments required when using a theorem, as most hypotheses can be inferred, and thus omitted.

The rest of the theorem is stated mathematically: if we know that `a` and `b` are both members of list `l`, then we can infer that `l` has at least two elements (`length l` returns a natural number representing the number of elements in the list). The → indicates a curried type, much like a curried function in functional programming languages.[1]

Lean allows for two kinds of proof: declarative and tactic. Figure 3.1 shows a tactic proof, which proceeds line-by-line through rewrite rules, axioms, and supporting lemmas to accomplish the goal of the theorem.

We start by adding the hypotheses `a ∈ L` and `b ∈ l` to the current context with `intros`. Before starting our proof, the context includes the hypotheses that `a` and `b` are elements of type α, etc. After assuming these hypotheses, our goal is to show that `2 ≤ length l`.

The proof then proceeds by casing on the form of `l`. Lists are defined inductively as either `nil` or as `cons x xs` of an element with the rest of the list (similar to `cons` from Lisp, Haskell, and OCaml). `cases` splits the goal into two goals, where in one goal, we assume that `l` is `nil`, and in the other, that `l` is `cons x xs`. In both branches, `l` is automatically replaced by the term of the appropriate form, both in the context and in the goal.

In the `nil` case, we find that our hypothesis `a ∈ L` has become `a ∈ nil`. Nothing can be a member of the empty list, as `not_mem_nil` states. We close our goal with `absurd`, which takes a hypothesis `p` and its negation `¬p` and provides the proof of any other proposition `q`.

In the `cons x xs` case, we case on the form of `xs`. In the first branch, we find that `a` and `b` are both equal to `x`, as they are members of a singleton list `[x]`. But that gives a contradiction with our assumption that `a ≠ b`. Contradictions close the proof branch just as with `absurd`.

In the second branch, we reduce our hypotheses using the rewrite `rw` tactic until we are left with a goal of `2 ≤ length ys + 2`. The theorem `le_add_self` closes the proof.

There are many more tactics available than those used in Figure 3.1. One commonly used tactic is `induction`, which invokes an induction rule associated with a type. For example, `induction n` when `n` is a natural number transforms the current goal into two: one where `n` is zero, and the other where `n` is the successor of another natural number and where an induction hypothesis is made available.

We refer interested readers to the learning page on the Lean community site for in-depth tutorials covering installation and setup and how to write tactic proofs.

[1] An instance of the Curry-Howard isomorphism [53].

I do not fear computers. I fear the lack of them.

Isaac Asimov

4 Library for verified encodings

In this chapter, we present the main features of our library for verified SAT encodings. All the proofs and instructions for how to compile them can be found at the following URL:

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

All files referenced in this work are hyperlinked to a frozen branch of the repository.

Our library is written in Lean 3 and makes liberal use of theorems from `mathlib` on natural numbers, lists, sets, and functions. We also depend upon Lean's simplifier to automate term rewriting and shorten proofs.

The library is roughly divided into three parts: (1) CNF representations, (2) supporting objects and functions, and (3) proofs of correctness. We discuss the former two here.

4.1 CNF representations

To formally verify theorems about Boolean formulas, we must first decide on a representation for them in Lean. Because many SAT solvers require formulas be in conjunctive normal form, we focus on a representation of CNF formulas.

To avoid using a specific type for Boolean variables (*e.g.* integers or strings), we use an arbitrary type `V`. Literals are either positive (`Pos v`) or negative (`Neg v`) forms of a variable `v`. Truth assignments are maps from the variable type `V` to a Boolean value.

Clauses and formulas are represented by lists of literals and clauses, respectively. One possibility would have been for them to be sets instead. Yet lists are a natural representation for clauses and formulas and have several benefits over sets. First, functions using lists in Lean's native functional programming language are computable, meaning that Lean's evaluator can produce real output; using sets would require more busywork to achieve the same end. Second, lists are a natural representation of DIMACS format, which specifies that formulas are lists of clauses, each of which is a list of literals. Third, many SAT solvers treat formulas as multisets—as lists—and so treating formulas as sets would not accurately capture how many solvers work [9].

The types for our CNF representations are summarized in Figure 4.1.

Our library defines and proves properties of operations on these types. We discuss a few notable ones here. The first is evaluation of clauses and formulas under an assignment. For example, the evaluation of a clause is implemented as the following:

```
inductive literal (V : Type*)
| Pos (v : V) : literal
| Neg (v : V) : literal

def clause (V : Type*) := list (literal V)
def cnf (V : Type*) := list (clause V)
def assignment (V : Type*) := V → bool
```

Figure 4.1: Types of clauses and CNF formulas in our library.

```
def eval (τ : assignment V) (c : clause V) : bool :=
    c.foldr (λ l b, b || (l.eval τ)) ff
```

In this definition, `foldr` is the usual fold operation on lists, and `l.eval τ` is the evaluation of a literal under τ. True and false in Lean are written as `tt` and `ff`, respectively. The evaluation of a CNF formula is written similarly. We chose built-in list operations over custom-written functions to shorten proofs with help from `mathlib`.

The expressions `c.foldr` and `l.eval` are instances of Lean's *anonymous projection* notation. Because `l` is inferred to have type `literal`, Lean 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`. The notation is convenient and we use it often.

We prove many theorems involving evaluation. A couple examples of useful theorems are below. The first theorem states that a clause evaluates to true only when there is a literal in it that evaluates to true. Its complement is that a clause evaluates to false only when all literals in the clause evaluate to false.

```
theorem eval_tt_iff_exists_literal_eval_tt {τ : assignment V} {c : clause V} :
  c.eval τ = tt ↔ ∃ (l : literal V), l ∈ c → l.eval τ = tt

theorem eval_ff_iff_forall_literal_eval_ff {τ : assignment V} {c : clause V} :
  c.eval τ = ff ↔ ∀ (l : literal V), l ∈ c → l.eval τ = ff
```

Other operations on clauses include counting the number of literals that evaluate to true under an assignment (`count_tt`), counting the number of literals in one clause which are negations of literals in another (`count_flips`), constructing a clause containing a list of variables such that it evaluates to false under an assignment (`falsify`), and constructing vars(·) for a clause (`vars`). The definitions for these operations are listed in Figure 4.2. We prove theorems about these operations as well.

All definitions of and proofs for these types can be found at `cnf/*.lean`. The proofs are more numerous and general than is required for our encoding correctness proofs, and so they stand as a basis for future verification efforts.

```
def count_tt (τ : assignment V) (c : clause V) : nat :=
    c.countp (literal.is_true τ)

def count_flips : clause V → clause V → nat
| []        _         := 0
| _         []        := 0
| (l :: ls) (m :: ms) := ite (l.flip = m)
                         (1 + count_flips ls ms) (count_flips ls ms)

def falsify (τ : assignment V) (l : list V) : clause V :=
    l.map (λ v, cond (τ v) (Neg v) (Pos v))

def vars : clause V → finset V
| []        := ∅
| (l :: ls) := {l.var} ∪ (vars ls)
```

Figure 4.2: Selected clause operations. `literal.is_true` returns true when a provided literal evaluates to true under τ. `l.flip` takes `Pos v` to `Neg v` and vice versa. `cond` and `ite` are conditional checks where `tt` gives the first branch and `ff`, the second.

4.2 Gensym

As mentioned previously, many encodings introduce additional auxiliary variables to reduce formula size. The process of fresh variable generation must be represented in our library for us to be able to define and prove the correctness of those kinds of encodings.

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 must be pedantic when we use Lean. We take inspiration from de Bruijn indices [19] and gensym objects [23] (such as in Lisp[1]) to create our own `gensym` object that generates fresh variables. We then prove that these variables don't collide with themselves or with variables already present in the formula. These proofs can be found at `cnf/gensym.lean`.

A `gensym` object is a pointer n on the natural number line and an injective function $f : \mathbb{N} \to \alpha$ for α arbitrary. The `gensym`'s pointer starts at $n = 0$. Requests for fresh variables can be made to the `gensym` via calls to `fresh`. `fresh` produces a new variable $f(n)$ and an updated `gensym` object with an incremented pointer $n := n + 1$. Because f is injective, then as n monotonically increases, the created variables $f(n)$ will all be distinct. We define the *stock* of a `gensym` to be the set of elements the `gensym` could produce. More formally, the stock S of a `gensym` g with offset n is

$$S(g) := \{x : \alpha \mid \exists\, d \in \mathbb{N},\ f(n + d) = x\}.$$

In the case that a `gensym` object should ignore a set of elements in its stock, the function `seed` can be used. When given a bijection $f : \mathbb{N} \leftrightarrow \alpha$ and a set X to avoid, `seed` creates a

[1] See `https://franz.com/support/documentation/10.1/ansicl/dictentr/gensym.htm`.

```
structure gensym (α : Type *) := (n : nat) (f : nat → α) (f_inj : injective f)

def init (f : nat → α) (f_inj : injective f) : gensym α :=
    { n := 0, f := f, f_inj := hf }

def fresh (g : gensym α) : (α × gensym α) := ⟨g.f (g.n), ⟨g.n + 1, g.f, g.f_inj⟩⟩

def nth (g : gensym α) (n : nat) : α := g.f (n + g.offset)

def stock : set α := { a | ∃ (n : nat), g.f (g.offset + n) = a }

def seed (f : nat → α) (f_inj : injective f) (fi : α → nat)
    (fi_inj : injective fi) (rinv : right_inverse f fi) : list α → gensym α
| [] := init hf
| l  := { n := 1 + max_nat (map fi l), f := f, f_inj := hf }
```

Figure 4.3: Gensym operations for fresh variable generation.

gensym where the offset n starts at one more than the maximum value $f^{-1}(x)$ attained by any $x \in X$. Since f is injective, setting $n := \max_{x \in X} f^{-1}(x) + 1$ makes $S(g) \cap X = \varnothing$.

fresh should be used when the definition of an encoding is recursive in nature: at any one function call, the needed variables can be generated, and then the new gensym can be passed to the recursive call. We use this strategy in the definition of an XOR encoding in Chapter 5. But when an encoding is defined without recursion, the gensym operation nth may be more appropriate. nth takes in a natural number i and returns the variable generated by the gensym after $i + 1$ calls to fresh.

The definitions of the gensym and its associated functions are listed in Figure 4.3. The function max_nat finds the maximum natural number when mapping elements in l under fi. Note that for seed, functions f and fi form the combined bijection for f. f is split into two functions to keep the function computable in Lean.

An equivalent definition for a gensym object would be to only track the injective function f. fresh would return an updated gensym where $f := (\lambda n,\ f(n+1))$. seed can be modified in a similar way to restrict the stock of the created gensym. We chose the offset representation because it was easier to reason about natural numbers than anonymous lambda functions.

In practice, encodings follow the DIMACS format. In encoding implementations, it is often the case that a global counter is used for variable generation. The counter is initialized to 1 (as DIMACS prevents any literals taking on the value 0). Whenever a new variable is required, the value of the counter is taken, and then the counter is incremented. In this way, the global counter represents a gensym object. The increment operation on the counter ensures the injectivity of fresh, and if the counter needs to avoid certain numbers, it can be initialized to a sufficiently large enough value, like what seed does.

When implementing encodings using gensym, an assumption that the input list of literals is disjoint from the stock of the gensym is required to ensure that fresh variables do not clash with variables already present. In practice, a global counter suffices.

```
def encodes (f : list bool → bool) (F : cnf V) (l : list (literal V)) :=
    ∀ τ, (f (l.map (literal.eval τ)) = tt ↔
    ∃ σ, F.eval σ = tt ∧ (τ ≡(clause.vars l)≡ σ))

def sequiv (F₁ F₂ : cnf V) (s : finset V) := ∀ τ,
    (∃ σ₁, F₁.eval σ₁ = tt ∧ (τ ≡s≡ σ₁)) ↔
    (∃ σ₂, F₂.eval σ₂ = tt ∧ (τ ≡s≡ σ₂))
```

Figure 4.4: The Lean representation of Definition 2.1. Note that because assignments are full maps $\tau : V \to B$, extension is handled via `eqod`, whose syntactic sugar is $\equiv \_ \equiv$.

4.3 Notions of encoding

The goal of our library is to prove that CNF formulas encode Boolean functions. In order to write such proofs, we require a representation of what an encoding is in Lean.

We express Definition 2.1 and Definition 2.2 in a straightforward manner. See Figure 4.4. Note that the input to encodings is a list of literals, but the input to Boolean functions is a list of Booleans. In the definition for `encodes`, we map the truth assignment τ on the list `l` to transform it into a list of Booleans.

The definition requires a notion of what it means to extend an assignment. We have chosen to make assignments maps from the variable type `V` to the boolean type `bool`. Thus, any assignment is a full assignment. To signify that two assignments share evaluations on a finite set of variables, we introduce a definition for `eqod`, short for equivalent-on-domain. The definition is

```
def eqod (τ₁ τ₂ : assignment V) (s : finset V) : Prop :=
    ∀ v ∈ s, τ₁ v = τ₂ v
```

A `finset` is a set type with a finite number of elements. To say, then, that an assignment σ extends τ, it suffices to specify that σ and τ agree on the variables of interest, and then that σ satisfies some other property, such as satisfying a CNF formula. The definition for `eqod` and associated theorems can be found at `cnf/assignment.lean`.

5 XOR encodings

The n-ary Boolean XOR function is commonly encountered in the translations of problems into SAT, such as in cryptography [40, 52], in approximate model counting [51], in the creation of matrix multiplication schemes [35, 43], and in the construction of set membership filters [58]. Many XOR encodings have been proposed [5, 29, 30, 43]. To the best of our knowledge, XOR encodings remain unverified. A lack of verification for these encodings means that, for example, analyses of cipher integrity with a SAT solver cannot be fully trusted, which introduces potential security risks in systems that use those ciphers. Our correctness proofs provide that trust.

In this chapter, we define the *direct* and *Tseitin* encodings for the XOR function, and then formally prove their correctness in Lean. The proofs mirror the pen-and-paper correctness proofs. However, the proof methods and supporting lemmas developed in Lean are more general and allow for easier implementation of future direct and recursive encodings of other Boolean relations and functions.

5.1 Encoding definitions

If x and y are Boolean values, then $x \oplus y = \top$ iff exactly one of x and y are true. We can extend this definition to include any number of Boolean values $x_1, \ldots, x_n$.

Definition 5.1 (Boolean XOR function)**.** *The* XOR *of n Boolean values is*

$$\text{XOR}(x_1, \ldots, x_n) := x_1 \oplus \cdots \oplus x_n = \top \textit{ iff an odd number of the } x_i \textit{ are } \top.$$

It is a simple matter to express XOR in terms of elementary propositional connectives, as

$$x \oplus y \equiv \neg(x \leftrightarrow y) \equiv (x \wedge \overline{y}) \vee (\overline{x} \wedge y) \equiv (x \vee y) \wedge (\overline{x} \vee \overline{y}).$$

The translation of XOR can be performed recursively, setting $x = x_1$ and $y = \text{XOR}(x_2, \ldots, x_n)$. However, many solvers would not accept such an input formula. A CNF encoding is needed.

The direct encoding

The first CNF encoding we will examine is the *direct* or *naive* encoding. The direct encoding is the most straightforward and introduces no additional literals, but also results in exponential blowup: the resulting formula has 2^{n-1} clauses. Such a formula quickly becomes intractable for solvers, so the direct encoding is not recommended above small n.

Definition 5.2 (Direct encoding of XOR)**.** *The direct encoding of the* XOR *function on literals* $x_1, \ldots, x_n$ *is*

$$\textsc{direct}(x_1, \ldots, x_n) = \bigwedge_{\textit{even number of negations}} \Big(\pm\, x_1 \vee \cdots \vee \pm x_n \Big).$$

Note that the negations are performed on an even number of literals, regardless if they are positive or negative, as the example shows.

Example 5.3. The direct encoding of $x_1 \oplus \overline{x}_2 \oplus x_3$ is

$$\textsc{direct}(x_1, \overline{x}_2, x_3) = (x_1 \vee \overline{x}_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3).$$

It is not hard to see that when τ satisfies $\textsc{direct}(x_1, \ldots, x_n)$, $\textsc{xor}(\tau(x_1), \ldots, \tau(x_n)) = \top$, and vice versa. In other words, the direct encoding is a CNF definition of XOR. To see why, consider such a satisfying τ. Because the direct encoding is in CNF, all its clauses must evaluate to true. An odd number of the x_i must evaluate to true under τ, since otherwise, the clause with exactly those even number of x_i negated in the direct encoding would evaluate to false, a contradiction. But that is precisely when XOR evaluates to true, by definition.

More generally, any Boolean function f has a direct CNF definition which is essentially a spelled-out truth table. Let $X := \{x_1, \ldots, x_n\}$ be the n Boolean inputs to f, let $S \subseteq X$ with $|S| := k$, and let τ be an assignment with $\text{vars}(\tau) = S$. Then if for all extended assignments σ from τ with $\text{vars}(\sigma) = X$, it is the case that $f(\sigma(x_1), \ldots, \sigma(x_n)) = \bot$, then the clause

$$C = \neg \left(\bigwedge_{i=1}^{k} x_{S_i}^{\tau} \right) \equiv \bigvee_{i=1}^{k} \overline{x}_{S_i}^{\tau}$$

is in the CNF definition. We mean $x^\tau = x$ when $\tau(x) = \top$ and $x^\tau = \overline{x}$ otherwise. In other words, for each subset of the variables in X with a partial assignment that cannot be extended to a full assignment that makes f true, a clause is added to the CNF definition disallowing the partial assignment.

For many common Boolean functions, such as XOR and pseudo-Boolean constraints, direct encodings are sometimes chosen for their simplicity. However, they often require a super-linear, or even an exponential, number of clauses, and so are not the preferred encoding above small n.

The Tseitin encoding

The direct encoding produces formulas which are too large for solvers above small n. More compact encodings are required for practical application. To that end, we turn to a common method for reducing the size of an encoding: the *Tseitin transformation* [55].

The Tseitin transformation is a general method of transforming arbitrary formulas in propositional logic into CNF such that the resulting formula is equisatisfiable with the original. Transformed formulas have the additional property of having length linear in the input formula size. By recursively introducing auxiliary literals via if-and-only-if relations with subformulas, the original formula can be rewritten without exponential blowup.

To get a sense of how the Tseitin transformation works, consider a propositional logic formula $F = F_1 \bullet F_2$, where $\bullet$ is an arbitrary binary Boolean connective. One round of the Tseitin transformation introduces a fresh literal y to produce the following formula:

$$T(F) = (F_1 \leftrightarrow y) \wedge (y \bullet F_2).$$

Depending on the sizes of F_1 and F_2, the transformation may recurse into the subformulas. It may also unfold the operator $\bullet$ into CNF: for example, if $\bullet = (\leftrightarrow)$, then $y \leftrightarrow F_2$ becomes $(\overline{y} \vee F_2) \wedge (y \vee \neg F_2)$. Because new literals are introduced via if-and-only-if relationships, the transformed formula $T(F)$ is equisatisfiable with F.

We can use the same strategy for $\textsc{xor}$. We first fix a *cutting number* $k \geq 3$ which determines how large our subformulas are. We then replace a $(k-1)$-literal portion of the $\textsc{xor}$ with a fresh literal and recurse. The transformation has length linear in n.

$$T_k(\textsc{xor}(x_1, \ldots, x_n)) = (\textsc{xor}(x_1, \ldots, x_{k-1}) \leftrightarrow y) \wedge (y \oplus T_k(\textsc{xor}(x_k, \ldots, x_n))) \qquad (k < n).$$

We note two things. First, $a \leftrightarrow b$ is equivalent to $a \oplus \overline{b}$, so we pull y into the $\textsc{xor}$ by writing $\textsc{xor}(x_1, \ldots, x_{k-1}) \leftrightarrow y$ as $\textsc{xor}(x_1, \ldots, x_{k-1}, \overline{y})$. Then, to produce a subformula in CNF on these k literals, we replace $\textsc{xor}(x_1, \ldots, x_{k-1}, \overline{y})$ with $\textsc{direct}(x_1, \ldots, x_{k-1}, \overline{y})$. If k is small, then the resulting number of clauses is not too large.

Second, the connective joining y and $T_k(\textsc{xor}(x_k, \ldots, x_n))$ is the same as the connectives joining $x_k, \ldots, x_n$. So as before, we move y into the subformula. Since the binary $\oplus$ connective is commutative, we have a choice of where to place y. One method is to place y in the leftmost position to read $T_k(\textsc{xor}(y, \ldots))$. Another method is to place y in the rightmost position to read $T_k(\textsc{xor}(\ldots, y))$. Encodings that use the former method are called *linear*; the latter, *pooled*. To see why, consider several steps of each method with $k = 3$ and with the direct encoding notated as $D(\ldots)$. The linear method produces the following formula:

$$\begin{aligned}
\textsc{linear}_3(x_1, \ldots, x_6) &= D(x_1, x_2, \overline{y}_1) \wedge \textsc{linear}_3(y_1, x_3, \ldots, x_6) \\
&= D(x_1, x_2, \overline{y}_1) \wedge D(y_1, x_3, \overline{y}_2) \wedge \textsc{linear}_3(y_2, x_4, x_5, x_6) \\
&= D(x_1, x_2, \overline{y}_1) \wedge D(y_1, x_3, \overline{y}_2) \wedge D(y_2, x_4, \overline{y}_3) \wedge D(y_3, x_5, x_6)
\end{aligned}$$

Skipping the intermediate steps, the pooled method produces the following formula:

$$\textsc{pooled}_3(x_1, \ldots, x_6) = D(x_1, x_2, \overline{y}_1) \wedge D(x_3, x_4, \overline{y}_2) \wedge D(x_5, x_6, \overline{y}_3) \wedge D(y_1, y_2, y_3).$$

Note how in the linear method, the y_i literals bookend each subformula, whereas in the pooled method, the y_i literals appear only in the rightmost position of each subformula and then are grouped all together in the final subformula. This difference may seem minor. However, it can have a big impact on solver performance [43]. For example, consider an assignment that satisfies the function $\oplus(x_1, \ldots, x_n)$ but falsifies the two clauses containing the literals x_1 and x_n in both encodings. The linear encoding would then require $O(n)$ updates to its y_i literals to make the formula evaluate to true, while the pooled encoding would only require $O(\log n)$ updates to its y_i literals.

The choice of cutting number is also critical for solver performance. When k is larger, each encoding introduces fewer auxiliary variables, but at a cost of larger direct encoding subformulas. Depending on the application, different cutting numbers are optimal. Soos and Meel argue for an optimal cutting number of $k = 4$ [51], while cutting numbers of $k = 6$ or $k = 7$ appear optimal for applications in cryptography [5, 43]. In our correctness proof, the cutting number is arbitrary.

We now formally define the encodings discussed above. We highlight the linear and pooled encodings in this work because they are the two most common Tseitin encodings of XOR.

Definition 5.4 (Tseitin transformations for XOR)**.** *Fix $k \geq 3$. Then the linear and pooled encodings for n literals are*

$$\text{LINEAR}_k(x_1, \dots, x_n) = \begin{cases} \text{DIRECT}(x_1, \dots, x_n) & n \leq k \\ \text{DIRECT}(x_1, \dots, x_{k-1}, \overline{y}) \wedge \text{LINEAR}_k(y, x_k, \dots, x_n) & o.w. \end{cases}$$

$$\text{POOLED}_k(x_1, \dots, x_n) = \begin{cases} \text{DIRECT}(x_1, \dots, x_n) & n \leq k \\ \text{DIRECT}(x_1, \dots, x_{k-1}, \overline{y}) \wedge \text{POOLED}_k(x_k, \dots, x_n, y) & o.w. \end{cases}$$

where in both cases, y is fresh.

Both the direct and Tseitin encodings encode the Boolean relation defined by the Boolean function $\text{XOR}(x_1, \dots, x_n) = \top$. To encode the relation defined by $\text{XOR}(x_1, \dots, x_n) = \bot$, one can instead encode $\text{XOR}(\top, x_1, \dots, x_n) = \top$ using either the direct or Tseitin encodings, as $\top \oplus x = \overline{x}$. Equivalently, one can encode $\text{XOR}(z, x_1, \dots, x_n) = \top$ and add the unit clause z.

The Tseitin transformation can also be used more generally to embed sub-encodings into formulas. For example, suppose the formula F was

$$F = \text{XOR}(x_1, \dots, x_n) \wedge (x_1 \vee z_1 \vee z_2) \wedge \dots.$$

Then a fresh variable y can be introduced and set in an iff relation with $\text{XOR}(x_1, \dots, x_n)$. Every instance of $\text{XOR}(x_1, \dots, x_n)$ can then be replaced by y. But as we saw above,

$$\text{XOR}(x_1, \dots, x_n) \leftrightarrow y \equiv \text{XOR}(x_1, \dots, x_n, \overline{y}),$$

so we may write

$$F \equiv (\text{XOR}(x_1, \dots, x_n, \overline{y})) \wedge y \wedge (x_1 \vee z_1 \vee z_2) \wedge \dots.$$

Then, one can choose one's favorite encoding for XOR (perhaps the linear encoding) as part of converting F into CNF. In this way, Boolean functions which are sub-constraints of a formula F can be encoded efficiently without affecting the other constraints. Similar strategies can be used for other Boolean functions, such as the at-most-one function.

5.2 Lean representations

We now present the Lean representations of the XOR function and the encodings from the previous section. We implement them in Lean's native functional programming language. Files for the definition of XOR and all XOR encodings can be found at `xor/*.lean`.

The XOR function is represented as a function that takes in a list of Boolean values and returns a single Boolean value. The evaluation of a list of literals under XOR and under an assignment first maps the assignment onto the literals of the list and then `foldr`s the result with the binary `bxor` operator. The characterization of the evaluation of XOR in terms of an odd number of literals evaluating to true is given by the theorem `eval_eq_bodd_count_tt`. Since the name "`xor`" was already defined in Lean 3, we chose `Xor` instead.

```
def Xor (l : list bool) : bool := l.foldr bxor ff

def Xor.eval (l : list (literal V)) (τ : assignment V) : bool :=
    Xor (l.map (literal.eval τ))

theorem eval_eq_bodd_count_tt (l : list (literal V)) (τ : assignment V) :
    Xor.eval τ l = bodd (clause.count_tt τ l)
```

The direct encoding

The direct encoding is defined as all clauses with an even number of negations on a list of literals. To implement it, we require a way to generate these 2^{n-1} clauses. We first define a sub-routine called `explode` which generates all possible (ordered) instances of positive and negative literals on an input list of variables. `explode` is loosely analogous to the powerset operator on sets, as it generates a list of 2^n lists. An example demonstrates this analogy:

Example 5.5. If $[x, y, z]$ is a list of three Boolean variables, then

$$\begin{aligned}\texttt{explode}([x,y,z]) = \ & [[x,y,z],[\overline{x},y,z],[x,\overline{y},z],[x,y,\overline{z}],\\ & [\overline{x},\overline{y},\overline{z}],[x,\overline{y},\overline{z}],[\overline{x},y,\overline{z}],[\overline{x},\overline{y},z]].\end{aligned}$$

We implement `explode` as a recursive function. The first variable in the list is added to the front of the recursive result twice: once as a positive literal, once as a negative literal. The positive and negative cases are appended together to form the final result. The definition of `explode` in Lean 3 is given here. It can be found in our library at `cnf/explode.lean`.

```
def explode : list V → list (clause V)
| []       := [[]]
| (v :: l) := (explode l).map (cons (Pos v)) ++ (explode l).map (cons (Neg v))
```

In our implementation of the direct encoding, we use `explode` to generate all possible negations on all literals but the first, `l`. Then for each clause in `explode`, the number of negations is compared against the input to `explode`. If there is an even number of negations, then `l` is added. Otherwise, `l` is flipped.

```
def direct_xor : list (literal V) → cnf V
| []        := [[]]
| (l :: ls) := (explode (map var ls)).map
    (λ c, ite (!bodd (c.count_flips ls)) (l :: c) (l.flip :: c))
```

`ite` is shorthand for the typical if-then-else, and `bodd` returns true when the natural number passed to it is odd, and false otherwise.

The Tseitin encoding

The Tseitin encoding is more complex than the direct encoding, as it introduces fresh variables. We present the linear encoding here as a recursive function, but a similar setup would work for any permutation in the recursive call.

We first fix a cutting number $k \geq 3$, as specified by the hypothesis `hk`. We then case on the length of the list `l`. If its length is at most the cutting number, the direct encoding is used. Otherwise, the first $k-1$ literals and the negated fresh literal are given to the direct encoding. The function recurses on the remaining $n-k+1$ literals and the fresh literal. The direct and recursive formulas are combined to form the final formula.

```
def linear_xor {k : nat} (hk : k ≥ 3) : list (literal V) → gensym V → cnf V
| l g := if length l ≤ k then direct_xor l else
          (direct_xor (l.take (k - 1) ++ [(Neg g.fresh.1)]))
        ++ (linear_xor ((Pos g.fresh.1) :: (l.drop (k - 1))) (g.fresh.2))
```

The `take` and `drop` functions return a list with only the first $k-1$ elements and a list without the first $k-1$ elements, respectively. The `.1` (`.2`) notation in `g.fresh.1` is shorthand for the first (second) element of a pair. `++` is shorthand for the `list.append` operation, which concatenates the elements of the first list with the second list.

In Lean 3, the definition of `linear_xor` requires additional justification to show that the computation terminates. Our proof can be found in the repository.

5.3 Proofs of correctness

We now present our proofs of correctness. All proofs follow the ones in our library.

Assume that $x_1, \ldots, x_n$ are n Boolean literals.

Theorem 5.6. *The* XOR *direct encoding is an encoding of* XOR*.*

Proof. We prove that the direct encoding satisfies Definition 2.1 for XOR on $x_1, \ldots, x_n$. In our library, the theorem can be found at `xor/direct_xor.lean` and is stated as:

```
theorem direct_xor_encodes_Xor (l : list (literal V)) :
    encodes Xor (direct_xor l) l
```

All other proofs of correctness use `encodes` in this manner.

Assume that τ is an assignment. Since the direct encoding introduces no new variables, it suffices to show that for any assignment τ on $x_1, \ldots, x_n$, τ satisfies the direct encoding iff $\text{XOR}(\tau(x_1), \ldots, \tau(x_n)) = \top$. Thus, we prove a stronger statement, stated in Lean as:

```
theorem eval_direct_xor_eq_eval_Xor (l : list (literal V)) (τ : assignment V) :
    (direct_xor l).eval τ = Xor.eval τ l :=
```

The proof follows the intuition given after Example 5.3.

The proof is trivial when $n = 0$, so assume that $n \geq 1$.

First, assume that $\text{XOR}(x_1, \ldots, x_n) = \top$. Then by `eval_eq_bodd_count_tt`, we know that an odd number of the x_i evaluate to true under τ. It suffices to show that every clause in `direct_xor l` evaluates to true under τ. Every clause in `direct_xor l` has an even number of negations compared to `l`. A supporting lemma tells us that because the number of negated literals in each clause (an even number) differs from the number of true x_i (an odd number), then the clause must evaluate to true under τ.

Now assume that $\textsc{xor}(x_1, \ldots, x_n) = \bot$. Then an even number of the x_i evaluate to true. It suffices to show that there exists a clause in `direct_xor l` that evaluates to false. Pick the clause which negates precisely the x_i that are true under τ. In our proof, `falsify` supplies that clause. Since an even number of x_i are negated in the clause, then it evaluates to false under τ, and the clause is in the formula, as the formula contains all clauses with an even number of negations. □

The proof of Theorem 5.6 follows what it means for a direct encoding to be a CNF definition of a Boolean function: clauses in the formula disallow assignments that would cause the Boolean function to evaluate to false. The general method is to case on the evaluation of the Boolean function. In the case where the function evaluates to true, then none of the clauses in the formula evaluate to false, because the assignment is not one of the disallowed ones. In the false case, then there is some violated constraint, which corresponds to a single clause in the CNF definition. It suffices to identify the clause in the formula that is falsified under the assignment.

We now turn to the correctness proofs for the Tseitin XOR encodings. In our library, the proofs can be found at `xor/tseitin_xor.lean`.

Theorem 5.7. *Let $k \geq 3$. Then $\textsc{linear}_k(x_1, \ldots, x_n)$ is an encoding of* XOR.

Proof. Fix $k \geq 3$ and let τ be a truth assignment on $\text{vars}(\{x_1, \ldots, x_n\})$. The proof is by strong induction on n. The proof is trivial when $n \leq k$, as the linear encoding becomes the direct encoding, which by Theorem 5.6 encodes XOR. So assume that $n > k$.

Let us now prove the forward direction: that if $\textsc{xor}(x_1, \ldots, x_n) = \top$ under τ, then there exists an extending assignment σ that satisfies $\textsc{linear}_k(x_1, \ldots, x_n)$. In Lean, the statement is written as:

```
lemma linear_forward (hk : k ≥ 3) (hdis : disjoint g.stock (clause.vars l)) :
  Xor.eval τ l = tt → ∃ (σ : assignment V),
  (linear_xor hk l g).eval σ = tt ∧ (τ≡(clause.vars l)≡σ)
```

We require the assumption that the `gensym`'s stock is disjoint from the variables in `l` to ensure that `g` produces fresh variables.

Assume that $\textsc{xor}(x_1, \ldots, x_n)$ is true under τ. Recall that the linear encoding introduces a single fresh variable y before recursing. We want to fix the truth value of y based on the truth values of the x_i so we can apply the induction hypothesis.

Case on the truth value of $\textsc{xor}(x_k, \ldots, x_n)$ under τ. Suppose that it evaluates to true. Then when we add y to $\textsc{xor}(y, x_k, \ldots, x_n)$, we don't want to change the overall truth value. Since $\bot \oplus x = x$ for any x, we see that we want y to be false. So we extend τ to a new assignment γ by setting $\gamma(y) = \bot$, where y is the first fresh variable generated by the `gensym g`. We next apply the induction hypothesis to find that $\textsc{linear}_k(y, x_k, \ldots, x_n) = \top$ under some other assignment σ, and that σ extends γ on $\text{vars}(\{y, x_k, \ldots, x_n\})$. Since σ does not extend τ on all the x_i, a combination of τ and σ' are used as the final witness. When a literal to be evaluated has its variable in $\text{vars}(\textsc{linear}_k(y, x_k, \ldots, x_n))$, then σ is used. Otherwise, τ is used. Some internal lemmas show that this combined assignment extends τ.

To finish the proof, it suffices to show that $\textsc{linear}_k(x_1, \ldots, x_n) = \top$ under this combined assignment. The recursive portion of the linear encoding is satisfied by the induction hypothesis. The direct encoding portion, $\textsc{direct}(x_1, \ldots, x_{k-1}, \overline{y})$, is satisfied because $\gamma(\overline{y}) = \top$ and by our assumption that $\textsc{xor}(x_k, \ldots, x_n) = \top$ under τ, then $\textsc{xor}(x_1, \ldots, x_{k-1}) = \bot$ under τ.

The proof is the same when $\textsc{xor}(x_k, \ldots, x_n) = \bot$ under τ, except with $\gamma(y) = \bot$ and with other parities flipped.

Let us now prove the reverse direction: that if there exists a satisfying assignment σ for $\textsc{linear}_k(x_1, \ldots, x_n)$ that extends τ, then $\textsc{xor}(x_1, \ldots, x_n) = \top$ under τ. Since σ and τ agree on the x_i, we prove the following statement in Lean:

```
lemma linear_reverse {k : nat} (hk : k ≥ 3) {l : list (literal V)} {g : gensym V}
    (hdis : disjoint g.stock (clause.vars l)) {σ : assignment V} :
    (linear_xor hk l g).eval σ = tt → Xor.eval l σ = tt
```

The statement is almost sufficient, but along with the fact that σ extends τ on the x_i, then $\textsc{xor}(x_1, \ldots, x_n) = \top$ under σ means the same under τ as well.

Assume that σ extends τ and satisfies $\textsc{linear}_k(x_1, \ldots, x_n)$. Let $y :=$ `g.fresh.1` be the first fresh variable generated by `gensym g`. Since the linear encoding evaluates to true under σ, then both $\textsc{direct}(x_1, \ldots, x_{k-1}, \overline{y})$ and $\textsc{linear}_k(y, x_k, \ldots, x_n)$ evaluate to true under σ. We invoke the induction hypothesis on the recursive linear encoding to derive that $\textsc{xor}(y, x_k, \ldots, x_n)\top$ under σ.

We now case on the value of $\sigma(y)$. If $\sigma(y) = \top$, then by `eval_eq_bodd_count_tt`, an even number of $x_k, \ldots, x_n$ evaluate to true under σ'. Similarly, an odd number of $x_1, \ldots, x_{k-1}$ evaluate to true under σ. So in total, an odd number of the x_i are true under σ (and thus τ), which occurs precisely when $\textsc{xor}(x_1, \ldots, x_n) = \top$, as we wished to show.

The proof is the same in the case that $\tau'(y) = \bot$, except with the parities of the evaluation of $x_1, \ldots, x_{k-1}$ and $x_k, \ldots, x_n$ under τ are flipped. □

Since the Tseitin encoding is recursive, the proof of Theorem 5.7 is by induction. The general method is to produce a satisfying assignment for the recursive formula, and then set the truth value of introduced variables to create the final truth assignment. Recursive encodings for other Boolean functions follow a similar form to the Tseitin encoding, and so their proofs of correctness should follow the one in Theorem 5.7.

Proof effort

We now discuss the time and lines of code (LOC) effort for the direct and Tseitin encodings for XOR.

The direct encoding is a simple encoding. Developing the supporting lemmas and completing the proof of correctness took two person-weeks.

The Tseitin encoding is more complex and required the use of `gensym`. It took about one person-week to write the `gensym` suite of theorems, and then two person-weeks to complete the proof of correctness. Note that some of the results used for the Tseitin encoding came from the direct encoding, so developing the Tseitin encoding from scratch would have taken at around three person-weeks.

File	LOC
gensym.lean	258
xor.lean	87
explode.lean	210
direct_xor.lean	223
tseitin_xor.lean	550

Table 5.1: Lines of code in files associated with XOR encodings.

The lines of code present in applicable files are listed in Table 5.1. Note that the LOC for the Tseitin encoding include a proof of correctness for both the linear and pooled encodings (the proofs are almost identical).

The greatest challenge for proving the direct encoding correct was developing mathematical machinery to tie the meaning of the Boolean function to the kinds of truth assignments each clause in the CNF definition disallowed. Once the connection to parity reasoning was made, the proof followed.

The greatest challenge for proving the Tseitin encoding correct was developing the right operations and supporting lemmas for the gensym. Not only do we require a hypothesis that the gensym's stock be disjoint from the variables in the list, but upon applying the induction hypothesis, the disjointness hypothesis needed to be updated. Additional work was required to stitch together the semi-extended assignment from the induction hypothesis with the rest of the variables.

6 At-most-one and at-most-k encodings

We saw in Section 2.2 that a reduction from the Hamiltonian cycle problem to SAT required the sub-encoding for EXACTLYONE. The EXACTLYONE constraint appears in many problems. For example, graph colorings can require that every vertex be colored with only one color, and scheduling problems can require that every timeslot be used exactly once. Thus, it is important to have a correct and efficient encoding for these applications.

One way of stating that exactly one literal should be true is that at least one (ALO) should be true and at most one (AMO) should be true. The encoding for ALO takes a single clause:

$$\text{ALO}(x_1, \ldots, x_n) = \bigvee_{i=1}^{n} x_i.$$

It is clear that to satisfy the clause, at least one of the x_i must evaluate to true.

The encoding for AMO is not as clear. Just as for XOR, many AMO encodings have been proposed [4, 24, 38, 44, 49], each with a different number of clauses and auxiliary variables.

In this chapter, we introduce two encodings for AMO: a *direct* encoding and the *Sinz* encoding [49]. We then prove them correct. We also introduce a generalization of the Sinz encoding for the at-most-k (AMK) function and prove it correct.

6.1 Encoding definitions

The at-most-one (AMO) constraint is an instance of a *cardinality constraint*, which specifies that among a set of inputs, a certain number of them should be true. Often, cardinality constraints take the form of an inequality:

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

where k is a fixed constant. We interpret $\tau(x_i) = \top$ as meaning $x_i = 1$ and 0 otherwise. For AMO, we take the left inequality and set $k = 1$.

Definition 6.1 (At-most-one function)**.** *The Boolean function* AMO *on n Boolean values $x_1, \ldots, x_n$ is*

$$\text{AMO}(x_1, \ldots, x_n) := \sum_{i=1}^{n} x_i \leq 1.$$

The direct encoding

A CNF definition of AMO disallows partial inputs which cannot be extended to a satisfying one. It is clear that when two distinct x_i are set to true in a partial input, then no full extended input can make AMO on the x_i evaluate to true. Thus, the direct encoding is a collection of binary clauses ensuring that for any distinct pair of inputs, it cannot be the case that both are true.

Definition 6.2 (Direct encoding of AMO)**.** *The direct encoding of the* AMO *function on Boolean literals* $x_1, \ldots, x_n$ *is*

$$\textsc{direct}(x_1, \ldots, x_n) = \bigwedge_{1 \le i < j \le n} (\overline{x}_i \vee \overline{x}_j).$$

The direct encoding produces $\binom{n}{2} \in O(n^2)$ clauses and introduces no auxiliary variables. Note that the negations of the literals are performed regardless of their polarity.

Example 6.3. The direct encoding of $\textsc{amo}(x_1, \overline{x}_2, x_3)$ is

$$\textsc{direct}(x_1, \overline{x}_2, x_3) = (\overline{x}_1 \vee x_2) \wedge (\overline{x}_1 \vee \overline{x}_3) \wedge (x_2 \vee \overline{x}_3).$$

The Sinz encoding

The Sinz encoding [49] is an encoding that requires $3n - 4 \in O(n)$ clauses and introduces $n - 1$ fresh *signal variables*. The idea is that when $x_i = \top$, any satisfying assignment must then set the corresponding signal variable $s_i = \top$. When an $s_i = \top$, it forces $s_{i+1} = \top$, propagating the signal, and it forces $x_{i+1} = \bot$, keeping the AMO constraint satisfied.

Definition 6.4 (The Sinz AMO encoding)**.** *The Sinz* AMO *encoding on Boolean literals* $x_1, \ldots, x_n$ *is*

$$\textsc{SinzAMO}(x_1, \ldots, x_n) = \left(\bigwedge_{i=1}^{n-1} (\overline{x}_i \vee s_i) \right) \wedge \left(\bigwedge_{i=1}^{n-1} (\overline{s}_i \vee \overline{x}_{i+1}) \right) \wedge \left(\bigwedge_{i=1}^{n-2} (\overline{s}_i \vee s_{i+1}) \right),$$

where the s_i *are fresh and pairwise distinct.*

We note that the bounds for the last collection of clauses $(\overline{s}_i \vee s_{i+1})$ run one fewer than the other two because x_n does not have a corresponding signal variable. Hence, there are $n - 1$ signal variables instead of n, and $3n - 4$ clauses instead of $3n - 3$.

Each binary clause in the Sinz encoding is of the form $(\overline{x} \vee y)$. The clause is logically equivalent to $x \Rightarrow y$, and thus accurately represents the idea that when x_i is true, then its signal variable s_i should be true (that $x_i \Rightarrow s_i$), and so on. Figure 6.1 shows how the encoding behaves under a truth assignment where $x_2 = \top$ and all other $x_i = \bot$.

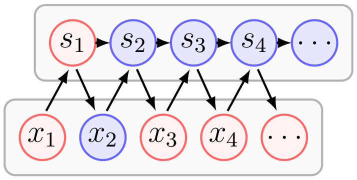

Figure 6.1: The Sinz AMO encoding under a satisfying truth assignment. Blue means the literal is true under the truth assignment, and red means the literal is false. Notice how the signal variables propagate that x_2 is true, enforcing that all later x_i must be false.

6.2 Lean representations

We present the Lean representation for the AMO function and the encodings presented in the previous section. All files can be found in our library at `cardinality/*.lean`.

The AMO function simply counts the number of true Boolean values in the list and compares it against 1. The list function `count` is from Lean's standard library and returns a natural number representing the number of elements in a list that equal a specified element. The `eval` function is almost identical to the one for XOR.

```
def amo (l : list bool) : bool := l.count tt ≤ 1

def eval (τ : assignment V) (l : list (literal V)) : bool :=
  amo (l.map (literal.eval τ))
```

The direct encoding

We implement the direct encoding recursively. While the list is nonempty, we split it into the first literal `l` and the remaining literals `ls`. We then take each literal in `ls` and create a binary clause with `l`. We return a CNF formula by appending the clauses containing `l` to the recursive result.

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

While the above implementation is recursive, a global definition would have sufficed. The two definitions are equivalent, especially when combined with supporting lemmas that relate the encoding to the `distinct` object that we defined:

```
def distinct {α : Type*} (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
```

`distinct` gives a shorthand to refer to two distinct elements in a list. Supporting lemmas allow us to relate to clauses in `direct_amo` using `distinct` without needing to always peer into the implementation, and so our correctness proof is somewhat implementation agnostic.

The Sinz encoding

We implement the Sinz AMO encoding non-recursively. We prefer a non-recursive implementation because we found it was easier to capture the global behavior of the signal variables than with the recursive encoding. We recommend using a non-recursive implementation for encodings which are not inherently recursive (as opposed to the Tseitin transformation). In the definition below, we define three helper sub-functions to generate each type of clause found in Definition 6.4.

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

def si_to_next_si (g : gensym V) (n i : nat) : cnf V :=
  if (i < n - 2) then [[Neg (g.nth i), Pos (g.nth (i + 1))]] else []

def si_to_next_xi (g : gensym V) (i : nat) (lit : literal V) : cnf V :=
  if (0 < i) then [[Neg (g.nth (i - 1)), lit.flip]] else []

def sinz_amo (l : list (literal V)) (g : gensym V) : cnf V :=
  if hl : length l < 2 then [] else
    join (map_with_index (λ (idx : nat) (lit : literal V),
      xi_to_si g (length l) idx lit ++
      si_to_next_si g (length l) idx ++
      si_to_next_xi g idx lit) l)
```

We use the list function `map_with_index`, which takes each element in a list and its index in the list (the first element has index 0, the second, 1, etc.) and transforms it under a given function $f : (\mathbb{N} \times \alpha) \to \beta$. We use the index to help determine if a clause should be generated at that index, as not every x_i has a corresponding signal variable.

The list function `join` takes each sub-formula generated by `map_with_index` and reduces the list of formulas to a single list. `join` is the same as `flat()` or `flatten()` in some other programming languages, such as Javascript.

Note that because our definition is non-recursive, we have to index into the `gensym` instead of passing along an updated `gensym`. We accomplish this by using the `nth` operation, rather than `fresh`. The injectivity of the function provided to the `gensym` ensures that as long as the indexes passed to `nth` differ, distinct fresh variables are produced.

While our definition is non-recursive, a recursive definition is possible. We did attempt a proof of correctness of a recursive definition first, but found it overly challenging. Our recursive definition was the following.

```
def sinz_rec : list (literal V) → gensym V → cnf V
| []                 _ := []
| [l_1]              g := []
| (l_1 :: l_2 :: ls) g := [l_1.flip, Pos (g.fresh.1)] ::
                          [Neg g.fresh.1, Pos g.fresh.2.fresh.1] ::
                          [Neg g.fresh.1, l_2.flip] ::
                          (sinz (l_2 :: ls) g.fresh.2)
```

The challenge of the correctness proof is that while the induction hypothesis gives back a satisfying assignment for `sinz (l₂ :: ls) g.fresh.2`, the truth values for the signal variables in the recursive formula depend on the truth value of the first signal variable. Thus, the truth value of the first signal variable affects all later signal variables. Contrast this with the proof of correctness for the linear encoding for XOR, which was free to set the truth value of the first fresh variable independent of what the other fresh variables received in the extension.

One way to salvage the recursive definition is to reverse the recursion: define the encoding on the first $n-1$ variables, and then extend it to include clauses on the last. However, we did not attempt this. Doing so may be an interesting line of future work.

6.3 Proofs of correctness

We now present our proofs of correctness. All proofs follow the ones in our library.

Assume that $x_1, \ldots, x_n$ are n Boolean literals.

Theorem 6.5. *The* AMO *direct encoding is an encoding of* AMO*.*

Proof. We prove that the direct encoding satisfies Definition 2.1 for AMO on $x_1, \ldots, x_n$. In our library, the theorem can be found at `cardinality/direct_amo.lean`. Assume that τ is a truth assignment. The proof is trivial when $n \leq 1$, so assume that $n \geq 2$.

For the forward direction, assume that $\text{AMO}(x_1, \ldots, x_n) = \top$. Then we know at most one of the x_i evaluate to true under $\top$. It suffices to show that all clauses in the direct encoding evaluate to true under τ. Pick any clause $C = (\overline{x}_i \vee \overline{x}_j)$ for distinct $i < j$. A supporting lemma `amo_eval_tt_iff_distinct_eval_ff_of_eval_tt` tells us that because $\text{AMO}(x_1, \ldots, x_n) = \top$, then if the first of any distinct pair of (x_i, x_j) is true, then the other must be false, else we would have a contradiction with our assumption that $\text{AMO}(x_1, \ldots, x_n) = \top$. The lemma (and the rest of our proof) makes use of the `distinct` object, which allows us to select two distinct elements from the list `l`.

So now, case on the value of $\tau(x_i)$. If $\tau(x_i) = \top$, then $\tau(x_j) = \bot$ by the lemma, and so $\tau(\overline{x}_j) = \top$, satisfying C. Otherwise, $\overline{x}_i$ satisfies C instead.

For the reverse direction, assume that σ is a truth assignment that satisfies the direct encoding and extends τ on the x_i. We want to show that $\text{AMO}(x_1, \ldots, x_n) = \top$ under τ. Since σ extends τ, it suffices to show that $\text{AMO}(x_1, \ldots, x_n) = \top$ under σ instead.

The same supporting lemma tells us that if we assume that x_i and x_j are distinct and $\sigma(x_i) = \top$, then we have to show that $\sigma(x_j) = \bot$. But this is direct from the fact that there is a clause in the direct encoding $(\overline{x}_i, \overline{x}_j)$ that is satisfied under σ. Because $\sigma(x_i) = \top$, that forces $\sigma(x_j) = \bot$. As our choice of distinct x_i and x_j was arbitrary, we have shown that for any distinct pair, the corresponding clause is satisfied, meaning that no two x_i are both true under σ. □

There are many similarities between Theorem 6.5 and Theorem 5.6, which tells us that there is a common proof method for proving direct encodings correct. Both proofs used supporting facts relating XOR and AMO being true to some condition on their inputs. For the former, XOR was true when an odd number of the x_i were true. For the latter, AMO was true when no distinct pair (x_i, x_j) were both true. Both allowed us to prove that no matter the clause in the formula, it was satisfied.

We now prove the Sinz encoding correct.

Theorem 6.6. *The Sinz encoding is an encoding of* AMO*.*

Proof. We prove that the Sinz encoding satisfies Definition 2.1 for AMO on $x_1, \ldots, x_n$. In our library, the theorem can be found at `cardinality/sinz_amo.lean`. Assume that τ is a truth assignment. The proof is trivial when $n \leq 1$, so assume that $n \geq 2$.

For the forward direction, assume that $\text{AMO}(x_1, \ldots, x_n) = \top$. Then at most one of the x_i evaluate to true under τ. Intuitively, we know that the truth values of the signal variables are determined by $x_1, \ldots, x_n$. If no inputs are true, then all signal variables can be false. Otherwise, if $\tau(x_i) = \top$, then any s_j with $j < i$ should be false, and any s_k with $k \geq i$ should be true. Providing this explicit extended assignment of τ would satisfy the direct encoding.

In Lean, we implement the assignment using a bit of machinery, which we present here.

```
def nth_from_var (g : gensym V) (v : V) : nat → nat
| 0       := 0
| (i + 1) := if v = g.nth (i + 1) then i + 1 else nth_from_var i

def signal_truth (l : list (literal V)) (τ : assignment V) : nat → bool
| 0       := l.head.eval τ
| (i + 1) := if h : (i + 1) < length l then
             (l.nth_le (i + 1) h).eval τ || signal_truth i
             else ff

def sinz_tau (l : list (literal V)) (g : gensym V) (τ : assignment V) :=
    λ v, if v ∈ clause.vars l then τ v else
         signal_truth l τ (nth_from_var g v (length l))
```

The assignment is called `sinz_tau`. When given a list `l`, a gensym `g`, and an assignment τ, it creates an extended assignment according to how the elements of `l` behave under τ. For variables `v` that are in $\text{vars}(\{x_1, \ldots, x_n\})$, the truth value of $\tau(v)$ is returned, so `sinz_tau` is a proper extension of τ. Otherwise, `sinz_tau` takes `v` and attempts to match it to a signal variable generated in the Sinz encoding. If $v = s_i$ for some i, then $(i - 1)$ is returned by `nth_from_var` (note the 0-indexing). Otherwise, we don't particularly mind what truth value `v` receives under `sinz_tau`, as it doesn't appear in the encoded formula.

Upon receiving such an i, the truth value of $v = s_i$ is determined by `signal_truth`. For our base case, when $i = 1$, then $s_1 = \tau(x_1)$. Otherwise, s_i is the logical OR of $\tau(x_i)$ and the truth value of s_{i-1} under our constructed extended assignment. Our explicit construction thus captures how signal variables should behave in the Sinz encoding.

Some lemmas help reduce `sinz_tau`, `signal_truth`, and `nth_from_var` into simpler forms when `v` is among the signal variables. We won't mention them here, but refer the interested reader to the proof in our library.

We claim that $\sigma :=$ `sinz_tau` satisfies $\text{SINZAMO}(x_1, \ldots, x_n)$. The proof is by casing on the form of any clause C, and then by casing on the truth values of the x_i and s_i involved in the clause. C is one of the three clause types as outlined in Definition 6.4. For each, we examine the two literals in the clause. We show that for each type of clause, `sinz_tau` does the appropriate thing. For example, suppose that $\tau(x_i) = \bot$ and $\sigma(s_{i-1}) = \top$. Then `sinz_tau` sets $\sigma(s_i) = \top$ in `signal_truth`, as we would expect. Setting $\sigma(s_i) = \top$ then satisfies the

$x_i \Rightarrow s_i$ and $s_{i-1} \Rightarrow s_i$ clauses. We prove that clauses of the form $s_i \Rightarrow x_{i+1}$ are true by using a supporting lemma that states what $\sigma(s_i) = \top$ only when at least one of $\{x_1, \ldots, x_i\}$ are true. Then, because we assumed that at most one of the x literals is true under τ (and thus σ), then it must be the case that x_{i+1} is false under τ.

For the reverse direction, assume there exists some σ' that extends τ and that satisfies $\textsc{SinzAMO}(x_1, \ldots, x_n)$. We use a lemma to show that $\sigma :=$ `sinz_tau` also satisfies the formula. We then use another lemma to show that if two distinct (x_i, x_j) are true under τ, then σ cannot satisfy the formula. Thus, at most one of the x_i must be true.

The lemma is a proof by induction on n that a signal variable s_i is true under σ only when one of $x_1, \ldots, x_i$ is true. Since that forces all following x_j to be false, we have our desired result. □

The key takeaway for this kind of proof is that for the forward direction, an explicit satisfying assignment is specified outside of the proof. In contrast, in Theorem 5.7, we produced an assignment inside the proof, based on the evaluation of XOR on subformulas. We recommend specifying an explicit satisfying assignment for non-recursive encoding implementations for use in the correctness proofs.

For the reverse direction, we used a lemma to tie semantic meaning to what $\sigma(s_i) = \top$ means. Once that was accomplished, an induction argument shows that the semantic meaning for each signal variable enforces that at most one x_i is true. Thus, we recommend relating semantic meaning to introduced variables using lemmas when the satisfying assignment is specified outside the proof of correctness.

Proof effort

We now discuss the time and lines of code (LOC) effort for these two encodings.

Once the `distinct` object had a sufficient number of supporting lemmas, the direct AMO encoding was even easier than the direct encoding for XOR. The proof of correctness took one person-week.

It is more difficult to gauge the difficulty of the Sinz AMO encoding. We initially tried a recursive definition for two person-weeks, but found that the proof was overly difficult. Finishing the proof of correctness for the non-recursive definition then took three person-weeks. So, in total, the process took five.

Table 6.1 shows the lines of code in files associated with the AMO encoding. Notably, the file `sinz_amo.lean` is the largest file in the library for an encoding at this point. The size of the file is due to the number of supporting lemmas in place to tie semantic meaning of the signal variables to the AMO function.

6.4 The Sinz at-most-k encoding

Since we only consider a single encoding of AMK, we compress the definition, Lean representation, and proof of correctness into a single section. Here, we prove correct the Sinz AMK encoding, a generalization of the Sinz AMO encoding.

File	LOC
amk.lean	308
alk.lean	127
distinct.lean	200
direct_amo.lean	282
sinz_amo.lean	754

Table 6.1: Lines of code in files associated with AMO encodings.

Definition 6.7 (At-most-k function)**.** *Let k be given. The Boolean function* AMK *on n Boolean values $x_1, \ldots, x_n$ for k is*

$$\textsc{amk}_k(x_1, \ldots, x_n) := \sum_{i=1}^{n} x_i \leq k.$$

Its Lean representation is analogous to AMO's, except with k a parameter.

```
def amk (k : nat) (l : list bool) : bool := l.count tt ≤ k

protected def eval (k : nat) (τ : assignment V) (l : list (literal V)) : bool :=
    amk k (l.map (literal.eval τ))
```

The Sinz AMK encoding has several slightly different forms. Shared across them is the idea that a $k \times n$ matrix of signal variables is generated. Using 1-indexing, the signal variable at the rth row and the cth column, $s_{r,c}$, is set to true when at least r of the $x_1, \ldots, x_c$ are true. A ternary clause that combines x_i and $s_{r,i-1}$ enforces that when both are true, $s_{r,i}$ is set to true.

In the encoding we consider, a $(k+1) \times n$ matrix is generated instead, but a unit clause enforces that $s_{k+1,n}$ be false. As a result, if at least $k+1$ of the x_i are true, then there is no satisfying assignment to the encoding.

Definition 6.8 (The Sinz at-most-k encoding)**.** *Let k be given. The Sinz at-most-k encoding on Boolean literals $x_1, \ldots, x_n$ is*

$$\begin{aligned}\textsc{SinzAMK}_k(x_1, \ldots, x_n) = &\left(\bigwedge_{i=1}^{n} (\overline{x}_i \vee s_{1,i})\right) \wedge \left(\bigwedge_{j=1}^{k+1} \bigwedge_{i=1}^{n-1} (\overline{s}_{j,i} \vee s_{j,i+1})\right) \\ &\wedge \left(\bigwedge_{j=1}^{k} \bigwedge_{i=1}^{n-1} (\overline{x}_{i+1} \vee \overline{s}_{j,i} \vee s_{j+1,i+1})\right) \wedge \overline{s}_{k+1,n},\end{aligned}$$

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

There are three types of clauses. The first kind, $(\overline{x}_i \vee s_{1,i})$, ensures that if x_i is true, then $s_{1,i}$, the signal variable for x_i on the first level of the matrix, is also true. The second kind, $(\overline{s}_{j,i} \vee s_{j,i+1})$, propagates a true signal variable on any row of the matrix to the right. We have already seen these two kinds of clauses in the Sinz AMO encoding.

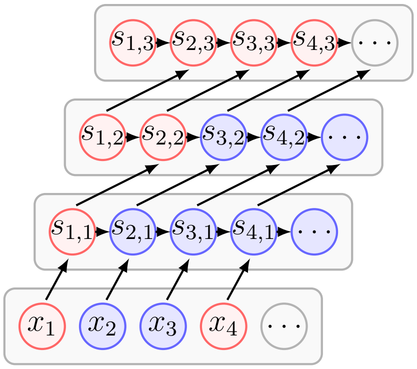

Figure 6.2: The Sinz AMK encoding under a truth assignment. Blue means that the literal is true under the assignment, and red means the literal is false. Notice how when a second x_i is true, the second row of signal variables propagates that two x_i are true. Note that the final negated unit clause is not shown.

The third kind, $(\overline{x}_{i+1} \vee \overline{s}_{j,i} \vee s_{j+1,i+1})$, ensures that if j of $x_1, \ldots, x_i$ are true and x_{i+1} is also true, then $s_{j+1,i+1}$ is true. In other words, whenever x_{i+1} is true, it sets the next row of the matrix true from that point on. Figure 6.2 depicts how the matrix of signal variables behaves under a given truth assignment.

So far, we have discussed an encoding of the Boolean relation defined by the Boolean function $\text{AMK}_k(x_1, \ldots, x_n) = \top$. The negated relation is the at-least-k function ALK. To encode the negated relation, $\text{ALK}_k(x_1, \ldots, x_n)$, one can encode $\text{AMK}_{n-k}(\overline{x}_1, \ldots, \overline{x}_n)$. The negation of each of the inputs ensures that if at most $n - k$ of the negated literals are true, then at least k of the original literals are true.

Because there is a two-dimensional indexing scheme for the signal variables, more machinery is needed to translate between a row and column in the matrix and the variable generated by the `gensym`. We present the Lean implementation here.

```
variables (g : gensym V) (l : list (literal V))

def S (r c : nat) : V := g.nth ((r * length l) + c)

def first_prop (r : nat) {c : nat} (hc : c < length l) :=
  ite (r = 0) [[(nth_le l c hc).flip, Pos (S g l 0 c)]] []

def signal_row_prop (r c : nat) :=
  ite (c > 0) [[Neg (S g l r (c - 1)), Pos (S g l r c)]] []

def inc_prop (r : nat) {c : nat} (hc : c < length l) :=
  ite (c > 0 ∧ r > 0) [[
    (nth_le l c hc).flip, Neg (S g l (r - 1) (c - 1)), Pos (S g l r c)
  ]] []

def neg_unit (k r c : nat) :=
  ite (r = k ∧ c = length l - 1) [[Neg (S g l r c)]] []
```

```
def sinz_amk (k : nat) : cnf V :=
if l = [] then [] else
join ((range (k + 1)).map (λ i, join
  ((range (length l)).map (λ j,
      signal_row_prop g l i j ++
      first_prop g l i j ++
      inc_prop g l i j ++
      neg_unit g l k i j )) ))
```

We discuss each definition in turn.

First, `variables` defines `g` and `l` so they don't have to be re-defined in each definition.

`S` defines the signal variable matrix. By providing a list `l` (thereby fixing n, the length of the list), `S` matches a row and column to the corresponding signal variable in the `gensym g`.

`first_prop`, `signal_row_prop`, and `inc_prop` generate the first, second, and third kinds of clauses from the encoding, respectively. `neg_unit` generates the unit clause on the final signal variable. Note that they generate CNF formulas with a single clause, which are appended together in the encoding.

`sinz_amk` puts all four kinds of clauses together, indexed by each row and column in the matrix. For rows and columns which are out of bounds for any particular clause generator, the clause generator ignores the input and outputs the empty formula, which is disregarded.

We now prove the encoding's correctness.

Theorem 6.9. *Let k be given. Then the Sinz* AMK *encoding is an encoding of* AMK.

Proof. The proof is analogous to one for the Sinz AMO, except the two-dimensional signal variable matrix requires more bookkeeping. □

Once again, the proof proceeds by associating a semantic meaning for when each signal variable is true.

Proof effort

The proof effort for the Sinz AMK function was very similar to the one for AMO. It took two person-weeks to implement the encoding and prove sufficient supporting lemmas. The file `sinz_amk.lean` is about 700 lines long, but a final supporting lemma needs to be finished, so the finished version will probably be about 1000 lines.

7 Conclusion

In this work, we examined the problem of encoding Boolean relations into SAT. We defined what it means for a Boolean formula F to encode a relation R, and then we proved particular encodings correct for XOR, AMO, and AMK. Our correctness proofs were presented in a library for verifying SAT encodings using the Lean theorem prover.

In addition to our correctness proofs, we identified several general methods of defining encodings and completing correctness proofs that will guide future work. For encodings which are recursive in nature, a recursive definition and induction proof is most natural for proving correctness. For encodings which are not recursive (imperative), a non-recursive definition is easiest to work with. Correctness depends on providing an explicit satisfying assignment, given some input assignment τ.

We also grappled with how to formalize the problem of generating fresh variables. We presented `gensym`. `gensym` supports operations which allow it to be used in both recursive and non-recursive implementations of encodings. By providing a hypothesis that a `gensym`'s stock is disjoint from literals already present, the `gensym` can provably provide fresh variables.

Future work

The library offers many lines of future work. Currently, our correctness proofs are for specific Boolean relations. Giljegård and Wennerbreck [27] verified general encoding methods, rather than encodings for single functions. One possible direction of interest for our library is verifying the Tseitin transformation correct, rather than instantiations of it.

We discussed in Section 4.1 that we represented clauses and formulas as lists in Lean. One natural extension for our library is to create a tool to translate a Lean representation of a formula to DIMACS format. That way, encodings that we verify in our library can produce output in DIMACS format, which can then be given to a SAT solver. Our library would then become a bridge between expressing encodings in Lean and running them on a solver.

One way to demonstrate the scope of the encodings in our library is to verify encodings of NP-hard problems correct. For example, Giljegård and Wennerbreck [27] proved their encoding of the n-queens problem was correct by using their correctness results for sub-encodings. We could do a similar verification of the naive encoding for the Hamiltonian cycle problem we discussed in Section 2.2. A more ambitious project would be to verify the encoding of the Keller conjecture [10].

There will also, of course, always be more encodings to prove correct (or even variations of the same encoding). Pseudo-Boolean and cardinality constraints, binary counters, and the order encoding are encodings of interest to verify next.

Another expansion in the scope of the library lies in verifying encodings more general than ones on Boolean relations. For example, our encoding described in Section 2.2 for the Hamiltonian cycle problem first took the graph problem and translated it onto a set of Boolean variables. While this inevitably must be done to encode it into Sat, the translation from graph problem to Boolean satisfiability problem is itself an encoding and requires verification. Expanding the definition of encoding to include more than Boolean relations would increase the range of results the library can provide.

Bibliography

[1] Mohammad Abdulaziz and Friedrich Kurz. Formally verified SAT-based AI planning, 2020.

[2] Khaza Anuarul Hoque, O. Ait Mohamed, Sa'ed Abed, and Mounir Boukadoum. An automated SAT encoding-verification approach for efficient model checking. In *2010 International Conference on Microelectronics*, pages 419–422, 2010.

[3] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Efficient generation of unsatisfiability proofs and cores in SAT. In *LPAR*, pages 16–30, 2008.

[4] Olivier Bailleux and Yacine Boufkhad. Efficient CNF encoding of boolean cardinality constraints. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming – CP 2003*, pages 108–122, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[5] Gregory V. Bard, Nicolas T. Courtois, and Chris 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.

[6] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

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

[8] Magnus Björk. Successful SAT encoding techniques. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:189–201, 07 2009.

[9] Jasmin Christian Blanchette, Mathias Fleury, and Christoph Weidenbach. A verified SAT solver framework with learn, forget, restart, and incrementality. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 4786–4790, 2017.

[10] Joshua Brakensiek, Marijn Heule, John Mackey, and David Narváez. The Resolution of Keller's Conjecture. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning*, pages 48–65, Cham, 2020. Springer International Publishing.

[11] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[12] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM.

[13] Fady Copty, Limor Fix, Ranan Fraer, Enrico Giunchiglia, Gila Kamhi, Armando Tacchella, and Moshe Y. Vardi. Benefits of bounded model checking at an industrial setting. In *CAV*, pages 436–453. Springer, 2001.

[14] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2):95–120, 1988.

[15] Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88*, pages 50–66, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[16] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt, Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In Leonardo de Moura, editor, *Automated Deduction – CADE 26*, pages 220–236, Cham, 2017. Springer International Publishing.

[17] Luís Cruz-Filipe, João Marques-Silva, and Peter Schneider-Kamp. Efficient certified resolution proof checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 10205, pages 118–135, 2017.

[18] Luís Cruz-Filipe, João Marques-Silva, and Peter Schneider-Kamp. Formally verifying the solution to the boolean pythagorean triples problem. *J. Autom. Reason.*, 63(3):695–722, 2019.

[19] N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972.

[20] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[21] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Conference on Automated Deduction (CADE) 2015*, pages 378–388. Springer, Berlin, 2015.

[22] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.

[23] Andrzej Filinski. Normalization by evaluation for the computational lambda-calculus. In *Proceedings of the 5th International Conference on Typed Lambda Calculi and Applications*, TLCA'01, pages 151—165, Berlin, Heidelberg, 2001. Springer-Verlag.

[24] Alan Frisch, Timothy Peugniez, Anthony Doggett, and Peter Nightingale. Solving non-boolean satisfiability problems with stochastic local search: A comparison of encodings. *Journal of Automated Reasoning*, 35:143–179, 01 2005.

[25] Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl. SAT solving for termination analysis with polynomial interpretations. In João Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing – SAT 2007*, pages 340–354, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[26] Ian P. Gent. Arc consistency in SAT. In F. van Harmelen, editor, *15th European Conference on Artificial Intelligence (ECAI 2002)*, pages 121–125. IOS Press, 2002.

[27] Sofia Giljegård and Johan Wennerbreck. Puzzle solving with proof. Master's thesis, Chalmers University of Technology, University of Gothenburg, 2021.

[28] Evguenii I. Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 10886–10891. IEEE, 2003.

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

[30] Matthew Gwynne and Oliver Kullmann. On SAT representations of XOR constraints. In Adrian-Horia Dediu, Carlos Martín-Vide, José Luis Sierra-Rodríguez, and Bianca Truthe, editors, *Language and Automata Theory and Applications - 8th International Conference, LATA 2014, Madrid, Spain, March 10-14, 2014. Proceedings*, volume 8370 of *Lecture Notes in Computer Science*, pages 409–420. Springer, 2014.

[31] Marijn J. H. Heule. Chinese Remainder encoding for Hamiltonian cycles. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing – SAT 2021*, pages 216–224, Cham, 2021. Springer International Publishing.

[32] Marijn J. H. Heule, Warren Hunt, Matt Kaufmann, and Nathan Wetzler. Efficient, verified checking of propositional proofs. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving*, pages 269–284, Cham, 2017. Springer International Publishing.

[33] Marijn J. H. Heule, Warren A. Hunt, Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 181–188. IEEE, 2013.

[34] Marijn J. H. Heule, Warren A. Hunt, Jr., and Nathan Wetzler. Verifying refutations with extended resolution. In *International Conference on Automated Deduction (CADE)*, volume 7898 of *LNAI*, pages 345–359. Springer, 2013.

[35] Marijn J. H. Heule, Manuel Kauers, and Martina Seidl. New ways to multiply 3×3-matrices. *J. Symb. Comput.*, 104:899–916, 2021.

[36] Marijn J. H. 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, *Theory and Applications of Satisfiability Testing – SAT 2016*, pages 228–245, Cham, 2016. Springer International Publishing.

[37] Franjo Ivančić, Zijiang Yang, Malay K. Ganai, Aarti Gupta, and Pranav Ashar. Efficient SAT-based bounded model checking for software verification. *Theoretical Computer Science*, 404(3):256–274, 2008.

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

[39] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 1–35, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[40] Anastasia Leventi-Peetz, Oliver Zendel, Werner Lennartz, and Kai Weber. CryptoMiniSat switches-optimization for solving cryptographic instances. In Daniel Le Berre and Matti Järvisalo, editors, *Proceedings of Pragmatics of SAT 2015 and 2018*, volume 59 of *EPiC Series in Computing*, pages 79–93. EasyChair, 2019.

[41] Filip Marić. Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. *Theoretical Computer Science*, 411(50):4333–4356, 2010.

[42] The mathlib community. The Lean mathematical library. In Jasmin Blanchette and Catalin Hritcu, editors, *Certified Programs and Proofs (CPP) 2020*, pages 367–381. ACM, 2020.

[43] Wojciech Nawrocki, Zhenjun Liu, Andreas Fröhlich, Marijn J. H. Heule, and Armin Biere. XOR local search for boolean brent equations. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*, volume 12831 of *Lecture Notes in Computer Science*, pages 417–435. Springer, 2021.

[44] Van-Hau Nguyen and Son 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, page 46–53, New York, NY, USA, 2015. Association for Computing Machinery.

[45] Van-Hau Nguyen, Van-Quyet Nguyen, Kyungbaek Kim, and Pedro Barahona. *Empirical Study on SAT-Encodings of the At-Most-One Constraint*, page 470–475. Association for Computing Machinery, New York, NY, USA, 2020.

[46] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[47] Natarajan Shankar and Marc Vaucher. The mechanical verification of a DPLL-based satisfiability solver. In Edward Hermann Haeusler and Luis Fariñas del Cerro, editors, *Proceedings of the Fifth Logical and Semantic Frameworks, with Applications Workshop, LSFA 2010, Natal, Brazil, August 31, 2010*, volume 269 of *Electronic Notes in Theoretical Computer Science*, pages 3–17. Elsevier, 2010.

[48] João P. Marques Silva and Inês Lynce. Towards robust CNF encodings of cardinality constraints. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pages 483–497. Springer, 2007.

[49] Carsten Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, volume 3709 of *LNCS*, pages 827–831. Springer, 2005.

[50] Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 28–32, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[51] Mate Soos and Kuldeep 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*, pages 1592–1599. AAAI Press, 2019.

[52] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009.

[53] Morten Heine B. Sørensen and Pawel Urzyczyn. Lectures on the Curry-Howard isomorphism, 1998.

[54] Yong Kiam Tan, Marijn J. H. Heule, and Magnus O. Myreen. cake_lpr: Verified propagation redundancy checking in CakeML. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *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*, volume 12652 of *Lecture Notes in Computer Science*, pages 223–241. Springer, 2021.

[55] Grigori S. Tseitin. On the complexity of derivation in propositional calculus. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 2*, pages 466–483. Springer, 1983.

[56] Allen Van Gelder. Verifying propositional unsatisfiability: Pitfalls to avoid. In João Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing – SAT 2007*, pages 328–333, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[57] Allen Van Gelder. Producing and verifying extremely large propositional refutations - have your cake and eat it too. *Ann. Math. Artif. Intell.*, 65(4):329–372, 2012.

[58] Sean A. Weaver, Hannah J. Roberts, and Michael J. Smith. XOR-satisfiability set membership filters. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *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*, volume 10929 of *Lecture Notes in Computer Science*, pages 401–418. Springer, 2018.

[59] Emre Yolcu, Scott Aaronson, and Marijn J. H. Heule. An automated approach to the Collatz Conjecture. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 468–484, Cham, 2021. Springer International Publishing.

[60] Neng-Fa Zhou. In Pursuit of an Efficient SAT Encoding for the Hamiltonian Cycle Problem. In *Principles and Practice of Constraint Programming: 26th International Conference, CP 2020, Louvain-La-Neuve, Belgium, September 7–11, 2020, Proceedings*, page 585–602, Berlin, Heidelberg, 2020. Springer-Verlag.