UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Tomaz Gomes Mascarenhas

Demonstrando teoremas em Lean por meio da reconstrução de provas em SMT

Belo Horizonte
2023

Tomaz Gomes Mascarenhas

Demonstrando teoremas em Lean por meio da reconstrução de provas em SMT

Versão Final

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais, como requisito parcial à obtenção do título de Mestre em Ciência da Computação.

Orientador: Haniel Barbosa

Belo Horizonte
2023

Tomaz Gomes Mascarenhas

Proving Lean theorems via reconstructed SMT proofs

Final Version

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Haniel Barbosa

Belo Horizonte
2023

"*Só quem sonha acordado vê o sol nascer.*"
(Unknown)

Resumo

Apesar de sua expressividade e robustez, assistentes de demonstração podem ser proibitivamente custosos para serem usados em formalizações de grande escala, dada a dificuldade de produzir as demonstrações interativamente. Atribuir a responsabilidade de demonstrar algumas das proposições a demonstradores automáticos de teoremas, como solucionadores de satisfatibilidade modulo teorias (SMT), é um jeito reconhecido de melhorar a usabilidade de assistentes de demonstração. Essa dissertação descreve uma nova integração entre o assistente de demonstração Lean 4 e o solucionador SMT cvc5.

Dada uma codificação de um teorema declarado em Lean como um problema de SMT e uma demonstração provida pelo cvc5 para o problema codificado, nós mostramos como traduzir essa demonstração para uma que certifique o teorema original em Lean. Para isso é necessário demonstrar a corretude, em Lean, dos passos lógicos tomados pelo solucionador. Desse modo, o verificador de demonstrações de Lean aceitará a demonstração em SMT do teorema original, caso o processo seja bem sucedido.

Essas técnicas são integradas no projeto Lean-SMT, que tem como objetivo criar uma tática em Lean que implemente o processo completo, isto é, a partir de um teorema em Lean, traduzi-lo para um problema formulado na linguagem do solucionador, invocar um solucinador para tentar resolvê-lo e produzir uma demonstração, e, caso ele seja bem-sucedido, traduzi-la para certificar o teorema original em Lean (o que é feito pelas técnias apresentadas aqui).

Palavras-chave: Verificação Formal, Lean, SMT

Abstract

Despite their expressivity and robustness, interactive theorem provers (ITPs) can be prohibitively costly to use in large-scale formalizations due to the burden of interactively proving goals. Discharging some of these goals via automatic theorem provers, such as satisfiability modulo theories (SMT) solvers, is a known way of improving the usability of ITPs. This thesis describes a novel integration between the ITP Lean 4 and the SMT solver cvc5.

Given the encoding of some Lean goal as an SMT problem and a proof from cvc5 of the encoded problem, we show how to lift this proof into a proof of the original goal. This requires proving the correctness, inside Lean, of the steps taken by the solver. Thus Lean's proof checker will accept the SMT proof as a proof of the original goal, in case this process is successful.

This set of techniques is part of the joint project Lean-SMT, which aims to create a tactic in Lean that implements the whole pipeline, that is, from a goal in Lean, translate it into a query in the solver's language, try to prove it using a solver and produce a proof and, in case it is successful, lift the proof produced, closing the original goal in Lean (which is done by the techniques presented here).

Keywords: Formal Verification, Lean, SMT

Glossary

$\overline{\boldsymbol{x}}$ Negation of the literal x.

$\boldsymbol{x \in C}$ Variable x is present in the clause C.

$\boldsymbol{C \setminus x}$ Clause C without variable x.

$\boldsymbol{Vars(\psi)}$ Set of all variables in the formula ψ.

$\boldsymbol{C_1 \diamond_x C_2}$ Clause resulting from applying resolution with C_1 and C_2 using x as a pivot.

$\boldsymbol{\psi_{\{x \leftarrow val\}}}$ Formula resulting of assigning the variable x to the boolean value val in ψ.

Contents

Chapter 1

Introduction

1.1 Context

A mechanized proof is a proof, written in some language recognized by a computer, that had its validity checked by a trusted verifier. One of the main applications of these artifacts are formalizing mathematical theories. Indeed, there are well-known examples of successful formalizations. One of them is the mechanization of the proof of a theorem regarding Perfectoid Spaces[11], done by the fields-medalist mathematician Peter Scholze, together with the community of a system called Lean[14]. Scholze proved the theorem using pen and paper, but was unsure of the result due to its complexity. Once he translated the theorem and the proof to the language of Lean, the system could point out some mistakes he made, and, after fixing them, he could be sure of the correctness of the proof.

Another application of mechanized proofs is verifying the correctness of mission-critical software. Given a specification of the behavior of some program, the program is said to be correct if it respects the specification for any input it is given. For instance, one could specify that a sorting routine must always produce the sorted permutation of it's input list. In this case, a given sorting routine is said to be correct if it indeed produces the desired permutation, regardless of which list it receives. There are a variety of techniques to obtain correctness evidence for a software. The most common one is the development of tests. Besides being easy to write an efficient set of tests, there are many types of bugs that can be discovered with its execution. In fact, this approach is enough for a large amount of problems that are solved by software engineering. However, tests can't guarantee that a program doesn't have flaws, since the number of valid inputs is almost always exceedingly large, or infinite. This kind of guarantee is extremely important for mission-critical software, that is, systems that have critical responsibilities, such as the control of airplanes or medical equipment. In this context, one promising alternative is to use a mechanized proof of the correctness of the software as an evidence for its safety.

The process of generating mechanized proofs can be divided into two categories: interactive and automatic.

Interactive theorem provers (ITPs) are mainly represented by proof assistants, in which, after defining a theorem, the user attempts to manually write a proof for it, relying on the tool to organize the set of hypothesis and how the goal changed step-wisely through the proof, as well as to ensure the correctness of each step according to a trusted kernel. In order to keep the kernel simple and small (and, therefore, easy to be trusted), it's implementation usually just straightforwardly checks the logic rules from the logic implemented by the ITP. Because of this, each step must be explicitly stated by the user, making the tool costly to be used.

Automatic theorem provers (ATPs), on the other hand, only require the user to define a conjecture, proceeding automatically to determine whether there exists a proof for it, or possibly providing a counter-example if it can find one. Although they are easier to use, ATPs require a large codebase to implement all the algorithms necessary to execute the search for a proof, making them more susceptible to errors and harder to be trusted. One possible way to overcome this trust issue is to produce a mechanized proof verifying the correctness of the ATP, however, besides being a very complex task, once the proof is done the development of the ATP becomes freezed, otherwise it would have to be verified again.

Another approach to increase the confidence in ATPs is to have them provide a proof to support their results, so that it can be independently verified whether it indeed proves the theorem in question. This has the downside of creating a need for allocating resources to verify the proof of every single theorem that is proved. On the other hand, as long as the proof format doesn't change, the implementation of the solver can be modified without requiring a modification in the checkers. Also, it is important to consider that it is often simpler to verify proofs than to verify the tool itself.

Another important advantage of the second approach is that it allows the ITPs to leverage the automatic proving performed by the ATPs by using the proofs they produce, since the requirement for accepting a proof, i.e. that each step is correct to its internal logic, can be applied to the ATP proof. By connecting these systems, it would be possible for the user of the ITP to focus on more complex steps of the proof, such as defining an induction hypothesis, while delegating the burden of other long and straightforward steps to the ATP. Indeed, this connection is so important that there are projects like Hammering Towards QED [7] that outline all the efforts that were already made in order to integrate interactive and automatic theorem provers. In this paper, the authors describe in detail each component that a system that creates a connection between ATPs and ITPs has to implement, as well as the main issues that they have to solve, based on existing programs that were successful in this task. Besides that, they show their potential through several large benchmarks.

1.2 Contributions

Given this context, we present a set of tools that would be an essential part of the integration between the ITP Lean 4 [14] and the SMT solver cvc5 [1]. Specifically, we aim to build a system that takes proofs of the unsatisfiability of SMT queries produced by cvc5 and reconstructs them in Lean. The main motivation of this project is that despite the fact that Lean is emerging as a promising programming language and proof assistant and being widely used by mathematicians in large-scale formalizations [17, 11], there is currently no way to interact with SMT solvers from it, even though these systems have been central in previous developments of proof automation in ITPs, as we will show in Sections 1.3.2 and 1.3.3. The contribution of the present work would enable a faster development of this kind of project using Lean.

We use the cvc5 solver because it already has a module for exporting proofs as Lean scripts [2], using a representation of the SMT terms[1] as an inductive type in Lean. However, these proofs are not fully verified by Lean's checker. Instead, a set of axioms are declared in Lean, representing all the logical rules that cvc5 uses to prove theorems, and the ITP only checks whether the rules were applied correctly and whether the end result of applying all the rules in the proof is, indeed, the required one. Our main contributions are to eliminate the need of increasing the trusted base by introducing those axioms and to make the proofs operate over native Lean terms, as opposed to terms of the inductive type that represents SMT terms.

Note that the set of tools we are proposing does not implement the full integration between Lean and cvc5. For instance, we do not implement a module for translating Lean goals into an equivalent SMT problem. However, our project is being used as part of the joint project Lean-SMT[2], that aims to implement a tactic in Lean that would perform the complete process, that is, starting from a Lean goal, translating it to a SMT query, invoking a solver to try to prove it and lifting the proof produced (in case it is found) to Lean's language, so that it can be used as a proof for the original goal.

[1] For more details about the SMT term language, see SMT-LIB [4].

[2] The code for the project can be found at https://github.com/ufmg-smite/lean-smt

1.3 Related Work

1.3.1 Hammering Towards QED

As previously mentioned, Hammering Towards QED is a project that aims to describe all the tools, which the paper calls "hammers", that were created with the purpose of connecting automatic and interactive theorem provers. Besides that, this document also outlines the main components that such tools usually have. They are the following:

- The premiss selection module: that identifies a subset of the facts previously demonstrated in the ITP that are more likely to be useful in order to prove the given goal, to be dispatched to the ATP.
- The translation module: that builds a problem in the language of the ATP that corresponds to the original goal from the ITP and using the premisses that were selected.
- The proof reconstruction module: that lifts the proof produced by the ATP into a proof that is accepted by the ITP.

In our case, we will restrict ourselves to implement a proof reconstruction module. The three main strategies used to reconstruct the proof produced by the automatic system inside the interactive one are also described in the paper.

The first one is to use the ITP to verify a deeply embedded version of the proof received, and, in case it is succesful, reflect this proof inside it's checker, proving the original goal. More specifically, the hammer defines a datatype to represent terms in the ATP and a set of functions to manipulate values on those datatypes, representing the axioms that the solver uses to transform the terms. Then, a lifting function is defined, that is, a function that take a value of this datatype and outputs an equivalent term in the native language of the ITP. Finally, the correctness of each transformation function is verified with respect to the lifting function, in the sense that, if the input term was lifted to a value that is provable in the ITP's logic, then the output term will also be provable. The ATP's proof will be represented as a sequence of those transformations, and their correctness are proved *a priori*. When checking a specific proof, the only step that the ITP must perform is to compute the result of the application of all the functions in the solver's output, and to check if the final term matches with the expected one. This technique is known as the Certified approach [9].

The second one is to match each axiom in the ATP's logic into a proved lemma or a tactic [is it okay to use tactic here? should I introduce the term first? –tom] defined in the ITP that works directly with native terms of the system. The proof produced by the automatic solver is then parsed into a sequence of applications of those lemmas and tactics and replayed inside the ITP. In this case, the proof is built on the fly and doesn't have it's correctness guaranteed (it can fail in the middle of the proccess in case the ATP or the hammer did something wrong). On the other hand, this technique skips computations done over embedded terms, which have to be done by the Certified approach, having the potential to have a better performance. This approach is known as the Certifying approach [9].

The third one is to compile the proof into the ITP's source code. This implies generating an actual script in the native language of the interactive system that corresponds to the proof received. After the script is generated, it is possible to postprocess[maybe copy the references that hammering towards use to talk about this? –tom] it in order to make the proof easier to be checked, in a way that the final script can possibly ignore a large portion of the original proof. This approach can be inconvenient for very large proofs, as it requires that the script is stored in some filesystem. However, it has the advantage over the two previous methods of only requiring access to the ATP on the first time that the proof is checked.

In this project we will be using the second approach. We give more details about this decision in the later chapters.

1.3.2 SMTCoq

One notable example of hammer is SMTCoq [15]. It is a plugin for the proof assistant Coq [6] that can be used as a tactic to prove theorems via their encoding into SMT and by lifting proofs produced by the SMT solvers veriT [10] and CVC4 [3]. The tool can support multiple solvers due to a preprocessor written in OCaml, that is able to turn the different proof formats they emmit into an unified certificate in the Coq language, that will be used as input for the plugin. Our tool explores one specific format of cvc5, therefore, for now, it only supports this solver and is less flexible in this aspect then SMTCoq.

The Coq hammer follows directly the Certified transformations approach, described in the previous section. It has a set of certified functions representing the transformations that can be done in the theory of Equality and Uninterpreted Functions and Linear Integer Arithmetic, as well as resolution chains. The way that SMT solvers prove that

a proposition is true is by showing that it's negation is unsatisfiable [it's okay to use unsatisfiable without defining it first? –tom]. This kind of proof is parsed in SMTCoq into a sequence of applications of certified functions, which have to transform the negation of the original goal into a term that will be lifted to the *False* proposition in Coq. Once the ITP verifies that the sequence of steps produced by the solver indeed produces *False*, the original theorem can be automatically closed, as we described in the previous section.

SMTCoq also differs from our work as it implements a translation module, but it doesn't implement a premiss selection module.

1.3.3 Sledgehammer

The ITP Isabelle/HOL [18] has a similar tool, namely, Sledgehammer [8]. This system have multiple strategies for building a proof for a theorem using ATPs. One of them is to invoke several ATPs in parallel with the given goal as their query and parse their output into a sequence of applications of Isabelle's predefined lemmas and tactics that could, in thesis, prove the original goal. This falls into the second category of strategies for proof reconstruction presented in Section 1.3.1.

Another technique used by Sledgehammer is to use the proof found by the ATP's only to scan which lemmas were used, and then discard it and build a whole new proof, feeding the lemmas found into existing plugins for Isabelle that already perform automatic theorem proving for the ITP. In this case, Sledgehammer works as a premiss selector (described in Section 1.3.1) for the other plugins.

1.4 Organization of this document

Chapter 2

Formal Preliminaries

2.1 Satisfiability Modulo Theories

2.1.1 Description of the Problem

The Boolean Satisfiability Problem (SAT) consists of, given a formula in Propositional Logic (PL) containing free variables, determine whether exists a function that assigns each variable to a boolean value, in a way that, after replacing the variables by their values provided by the function, the formula is evaluated to *true*. We say that a formula is satisfiable if such function exists, and unsatisfiable otherwise. In this work, we will be focusing on the problem CNF-SAT, an equivalent version of SAT in which the input formula always comes in Conjuctive Normal Form, that is, a conjunction of disjunctions. From now on, we will use SAT to refer to the CNF-SAT problem. Besides that, we will use the name *clause* to refer to one of the disjunctions in some input formula for SAT.

Satisfiability Modulo Theories (SMT) [5] is a generalization of SAT. There are two additions in this version of the problem: the first one is that the input formula can contain quantifiers binding variables, which will affect the satisfiability of that formula. Therefore, any formula in First Order Logic is a valid input to the SMT problem. The second addition is the inclusion of a set of theories that allows the problem to refer to variables of different domains. More precisely, a theory consists of a sort (for instance, integers) over which a subset of the variables of the problem can range over and a set of symbols that represents operations over these sorts with predefined semantics (for instance, addition and comparison operations). SMT problems are allowed to use multiple theories in the same instance. The logic framework that corresponds to Propositional Logic with these two additions is known as Many-Sorted First Order Logic (MSFOL). The semantics of this logic is given in detail in [16]. In Section 2.1.2 we give a brief overview of it.

2.1.2 Many-Sorted First Order Logic

define formarly many sorted first order logic

talk about fixing the meaning of some theories to improve performance

skipping this for now, I will write the rest of the thesis so I have a better understanding of what needs to go here

2.1.3 Applications

Given a program and a formal specification of some property related to the program, it is often [always? –tom] the case that we can express the proposition that asserts that the program satisfy the property as an SMT instance. For instance, consider the well known *abs* function, that takes an integer and returns its absolute value. The usual way to implement it is through a branch that checks whether the input variable is positive or negative. In case it is positive, its own value is returned. Otherwise, the value multiplied by -1 is returned:

Algorithm 1 Original Absolute Function

```
function ABS(x)
    if x < 0 then
        return −x
    else
        return x
    end if
end function
```

In program analysis, it is quite useful to eliminate branches from programs since this action completely removes one possible path that the flow of the program can take (besides optimizing it's performance). Obviously, this operation must be done with caution to not modify the original behavior of the program. In his book [20], Henry Warren proposes an alternative implementation for the *abs* function which doesn't have branches:

Where $\bigoplus$ represents the bitwise xor operation. We can design an instance of the SMT problem that asserts that both implementations produce the same output, when given the same input. We present the instance written in SMT-LIB [4], a standardized syntax for representing SMT problems:

Algorithm 2 Branchless Absolute Function

function ABS'(x)
  $y \leftarrow x >> 31$
  return $(x \bigoplus y) - y$
end function

```
1  (set-logic QF_BV)
2  (declare-const x (_ BitVec 32))
3  (declare-const result1 (_ BitVec 32))
4  (assert (= result1 (ite (bvslt x #x00000000) (bvneg x) x)))
5  (declare-const y (_ BitVec 32))
6  (declare-const result2 (_ BitVec 32))
7  (assert (= y (bvashr x (_ bv31 32))))
8  (assert (= result2 (bvsub (bvxor x y) y)))
9  (assert (distinct result1 result2))
10 (check-sat)
```

First, we set the combination of theories that will be used in this problem. In our case, we will be using *QF_BV*, which stands for quantifier free bitvectors. This means that this instance of the problem is not allowed to use quantifiers and is allowed to declare and use variables living in the Bitvector sort, as well as operations over this sort. Bitvectors are fixed-length arrays of bits. They are useful for representing machine integers, as they can simulate their semantics.

Next, in lines 2 and 3, we define two constants, both from the sort *BitVec 32* (arrays of 32 bits): x and *result1*. The first one represents the input value from the original *abs* function, and the second one, the result produced by that function. We then have to add an assertion in line 4 that binds the variable *result1* to the output of the function *abs* in terms of x. We translate the branch from the pseudocode as the *ite* operator, the comparison as the *bvslt* operator and the multiplication by -1 as the *bvneg* operation. This assertion will be encoded as a regular implication from SAT, where the premiss is it's proposition and the conclusion is the rest of the problem, which still needs to be defined. In lines 5 to 8 we repeat the process to define y and *result2*, which corresponds to the result of the branchless *abs* function. Finally, we assert that *result1* and *result2* must be different in line 9. If this problem is satisfiable, then there is a value for x which produces different values in each function. If it is unsatisfiable, then we can be sure that no such value exists, therefore, both functions are equivalent. Note that we are not verifying that the actual code of the functions are equivalent, just an abstraction over it's implementation.

2.1.4 SMT Solvers

A SMT solver is a piece of software whose main goal is to solve the SMT problem. Many-Sorted First Order Logic is undecidable in it's most general form [19], therefore SMT solvers have to limit themselves to use heuristics to solve the largest possible subset of instances of the problem. In this section we present the ideas used by these systems that are most relevant to the present work.

DPLL

First, let's explore how SAT is solved. Although MSFOL is not decidable, Propositional Logic is, therefore, it is possible to design a decision procedure for SAT. Indeed, one simple way to check whether a formula in PL with n variables is satisfiable or not is to simply test each one of the 2^n functions assigning truth values to those variables.

A more efficient alternative of a decision procedure for PL is the DPLL algorithm [12]. DPLL is based on the *Resolution* theorem:

Theorem 1 (Resolution) *Let x be a literal. Let C_1 and C_2 be two clauses such that $x \in C_1$ and $\overline{x} \in C_2$. Then $C_1 \wedge C_2 \rightarrow (C_1 \setminus x) \vee (C_2 \setminus \overline{x})$.*

More specifically, it is based on *Unit Resolution* (UR), that is, a more specific version of Resolution in which $C_1 = \{x\}$ or $C_2 = \{\overline{x}\}$. We present DPLL's pseudocode and a brief explanation over it:

Algorithm 3 DPLL Algorithm

Input: ψ, a formula in CNF
Output: *true* or *false*, depending whether ψ is satisfiable

```
function DPLL(ψ)
    if ∃C ∈ ψ. C = {⊥} then
        return false
    else if ∀C ∈ ψ. ⊤ ∈ C then
        return true
    else
        if ∃x ∈ Vars(ψ) such that x is a target for UR then
            ⟨C1, C2⟩ ← findClauses(x, ψ)  ▷ Clauses suitable for applying UR with x
            return DPLL(ψ ∪ C1 ⋄x C2)
        else
            Let x be an unassinged variable in ψ
            return DPLL(ψ{x←⊤}) ∨ DPLL(ψ{x←⊥})
        end if
    end if
end function
```

The algorithm works as follows: first, it checks to see if the formula can be evaluated to *true* or *false*. In case it can't, the procedure finds as many variables in which it can apply Unit Resolution as possible, calling itself recursively with each new application found. By the resolution theorem, the formula that will be used as a parameter in the recursive call is satisfiable if and only if the one that was received by input is also satisfiable, therefore, this step is sound. Once there are no more possibilities, it chooses an arbitrary variable and make two recursive calls: one assigning this variable to *true* and the other one to *false*. Since these are the only two possibilities for that variable, the input formula is satisfiable if and only if one of the recursive calls returned *true*. The algorithm uses this information to correctly return the disjunction between the two return values.

The actual algorithm used by most SMT solvers is a refinement over DPLL, called CDCL [13]. Since this refinement is not relevant for this work, we will not present it here.

DPLL(T)

DPLL(T) is a framework based on the DPLL algorithm that is used as a basis for solving the SMT problem. This is the solution that most SMT solvers actually implement for solving the problem. Consider a formula ψ over a theory $\mathcal{T}$ in MSFOL. Let's assume we have a solver for this theory (that is, a method for deciding whether a given set of propositions in $\mathcal{T}$ is consistent or not). The idea behind DPLL(T) is to create a PL formula ψ' from ψ by substituting each atom in it for a fresh boolean variable. We can then use the previously described DPLL algorithm to determine whether ψ' is satisfiable. If it is unsatisfiable, then ψ is also unsatisfiable. Otherwise, we can find a model M for ψ'. Although M satisfies ψ', it is possible that it contradicts some fact about $\mathcal{T}$. For instance, consider ψ to be the formula $x > 3 \wedge x < -2$. From it, we would generate the PL formula $p \wedge q$, where p represents $x > 3$ and q represents $x < -2$. The only possible model for this formula is the one that assigns both p and q to $\top$, but this is not valid when we translate back to ψ, as x cannot be both greater than 3 and smaller than -2. If this happens, we rely on the theory solver to provide a new lemma from $\mathcal{T}$ that shows why the previous assignment was invalid. In this case, it would provide the lemma $\neg(p \wedge q)$.

Algorithm 4 DPLL(T) Algorithm

Input: ψ, a formula in MSFOL over a theory $\mathcal{T}$
Output: *true* or *false*, depending whether ψ is satisfiable
function DPLL(T)(ψ)
end function

Congruence Closure

- Dar a definicao formal de EUF
- Explicar o que eh o congruence closure

- Explicar o algoritmo com union find pra resolver

Linear Arithmetic

2.1.5 Proofs in cvc5

2.2 Lean

take a look at chapter 2 of smtcoq

- Falar sobre porque eh facil confiar no proof assistant
- Explicar que taticas extendem a linguagem mas nao aumentam o trusted core

2.3 Lean's Framework for Metaprogramming

Chapter 3

Certifying Reconstruction of SMT Proofs in Lean

3.1 Certified vs Certifying

3.2 Classical vs Intuitionist (?)

3.3 Tactics

3.4 The Complete Architecture

3.5 Skipping the Parser

Chapter 4

Evaluation

Chapter 5

Future Work

Bibliography

[1] Haniel Barbosa et al. cvc5: A versatile and industrial-strength smt solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442, Cham, 2022. Springer International Publishing.

[2] Haniel Barbosa, Andrew Reynolds, Gereon Kremer, Hanna Lachnitt, Aina Niemetz, Andres Notzli, Alex Ozdemir, Mathias Preiner, Arjun Viswanathan, Scott Viteri, Yoni Zohar, Cesare Tinelli, and Clark Barrett. Flexible proof production in an industrial-strength smt solver. In Jasmin Blanchette, Laura Kovacs, and Dirk Pattinson, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, Lecture Notes in Computer Science. Springer, 2022.

[3] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification (CAV)*, pages 171–177. Springer, 2011.

[4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at www.SMT-LIB.org.

[5] Clark Barrett and Cesare Tinelli. *Satisfiability modulo theories*. Springer, 2018.

[6] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[7] Jasmin C. Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. Hammering towards qed. *Journal of Formalized Reasoning*, 9(1):101–148, Jan. 2016.

[8] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending sledgehammer with smt solvers. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23*, pages 116–130, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[9] Valentin Blot, Louise Dubois de Prisque, Chantal Keller, and Pierre Vial. General automation in coq through modular transformations. *Electronic Proceedings in Theoretical Computer Science*, 336:24–39, jul 2021.

[10] Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. veriT: An Open, Trustable and Efficient SMT-Solver. In Renate A. Schmidt, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2009.

[11] Davide Castelvecchi. Mathematicians welcome computer-assisted proof in "grand unification" theory. *Nature*, 595, 06 2021.

[12] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[13] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, jul 1960.

[14] Leonardo 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, *Automated Deduction - CADE-25*, pages 378–388, Cham, 2015. Springer International Publishing.

[15] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark Barrett. Smtcoq: A plug-in for integrating smt solvers into coq. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, pages 126–133, Cham, 2017. Springer International Publishing.

[16] María Manzano. *Extensions of First Order Logic*. Cambridge University Press, USA, 1996.

[17] The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages 367–381, New York, NY, USA, 2020. Association for Computing Machinery.

[18] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[19] Anderson Nakano Timm Lampert. Explaining the undecidability of first-order logic. 2021.

[20] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Professional, 2nd edition, 2012.