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 demonstrações em SMT

Belo Horizonte
2023

Tomaz Gomes Mascarenhas

Demonstrando teoremas em Lean por meio da reconstrução de demonstrações 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 Moreira 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 Moreira Barbosa

Belo Horizonte
2023

Mascarenhas, Tomaz Gomes.

M395d Demonstrando teoremas em Lean por meio da reconstrução de demonstrações em SMT [recurso eletrônico] / Tomaz Gomes Mascarenhas - 2023.

1 recurso online (82 f. il., color.) : pdf.

Orientador: Haniel Moreira Barbosa.
Dissertação (Mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciências da Computação.
Referências: f. 78-81

1. Computação – Teses. 2. Teoria da computação – Teses. 3. Engenharia de software - Teses. 4. Métodos formais- Teses. I. Barbosa, Haniel Moreira. II. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Computação. III.Título.

CDU 519.6*32(043)

Ficha catalográfica elaborada pela bibliotecária Irénquer Vismeg Lucas Cruz
CRB 6/819 - Universidade Federal de Minas Gerais - ICEx

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

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

TOMAZ GOMES MASCARENHAS

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

Haniel M. Barbosa

Prof. Haniel Moreira Barbosa - Orientador
Departamento de Ciência da Computação - UFMG

Fernando M Q Pereira

Prof. Fernando Magno Quintão Pereira
Departamento de Ciência da Computação - UFMG

Jeremy Avigad

Prof. Jeremy Avigad
Department of Philosophy - Carnegie Mellon University

Belo Horizonte, 12 de setembro de 2023.

Agradecimentos

Em primeiro lugar, eu agradeço ao Haniel, que, além de ser meu orientador e tornar tudo isso possível, se tornou um valioso amigo durante o mestrado. Haniel teve uma gentileza e eficiência excepcional para me guiar no processo. Eu devo muito a ele.

Eu agradeço também aos membros da banca, Fernando e Jeremy. Os dois são cientistas incríveis que tiveram a paciência de ler e entender a dissertação e prover uma perspectiva muito relevante sobre ela.

Nos últimos dois anos eu tive a oportunidade de conhecer e trabalhar com muitas pessoas interessantes. Em particular: Leonardo de Moura, Bruno Andreotti, Arjun Viswanathan, Hanna Lachnitt e todas as pessoas do time do cvc5. Sou grato por todas as conversas e pelo impacto positivo que tiveram em mim.

Eu fui introduzido à teoria dos tipos e verificação formal pelo Maurício Collares, um ex-professor da UFMG. Maurício é uma pessoa fantástica, com uma inteligência e gentileza fora do comum. Me sinto privilegiado por tê-lo conhecido e por todas as reuniões aprendendo Agda com ele.

Agradeço também aos meus pais, Fábio e Telma, por me criarem e pelo amor e suporte incondicional. As circunstâncias da vida têm tornado difícil o nosso encontro. Agradeço pela resiliência e compreensão deles para passar por esse desafio.

Ao longo da vida eu tive a sorte de criar amizades que me ajudaram a me manter são nesse mundo. Quero agradecer especialmente ao João, Higor, Augusto, Matheus, Victor, Daniel e a Sylvia, que, na medida do possível, se manteram próximos nos últimos anos e são muito importantes para mim. Sou feliz por ter conhecido cada um deles.

Por fim, eu gostaria de agradecer a Lorena, a pessoa que mais me conhece no mundo. Por mais de 6 anos de amor, amizade, carinho, cumplicidade, risadas e aventuras, eu serei eternamente grato.

“*The road to wisdom? Well, it’s plain and simple to express: err and err and err again,*
but less and less and less.”
(Piet Hein)

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 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, caso o processo seja bem sucedido, o verificador de demonstrações de Lean aceitará a demonstração SMT dada pelo solucionador para o teorema original.

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 SMT, invocar um solucinador para tentar resolvê-lo e produzir uma demonstração, e, caso ele seja bem-sucedido, tentar 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. This will allow Lean's proof checker to accept the SMT proof as a proof of the original goal.

This set of techniques is part of the joint project Lean-SMT, which aims to create a tactic in Lean that can soundly invoke an external SMT solver by translating a Lean goal into an SMT query; proving the goal using a proof-producing SMT solver; and, when successful, lifting the SMT proof to close the original Lean goal (which is done by the techniques presented here).

Keywords: Formal Verification, Lean, SMT

Contents

Chapter 1

Introduction

A mechanized proof is a proof written in a language recognized by a computer, so that its validity can be checked by a verifier. An important application of these artifacts is the formalization of mathematical theories. Indeed, there are well-known examples of successful formalizations [27, 28]. One of them is the mechanization of the proof of a theorem regarding Perfectoid Spaces [18], done by the fields-medalist mathematician Peter Scholze together with the community of a system called Lean [23]. Scholze proved the theorem using pen and paper, but was unsure of the result due to its complexity. With the help of Lean's community, he could translate the theorem and the proof to the language of Lean. Once the rewrite was complete, the system could find errors in the proof, and, after addressing all of them and obtaining confirmation from Lean's checker that the proof was correct, he could be sure of the result.

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 it for any given input. For instance, one could specify that a sorting routine must always produce the sorted permutation of its input list. In this case, a given sorting routine is said to be correct if it produces the desired permutation, for every 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. Tests are easy to be written and effective to find errors in programs. In fact, this approach is enough for a large amount of problems that are solved by software engineering. However, tests cannot guarantee that a program does not have flaws in general, 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.

Systems whose objective is to assist humans in the production of mechanized proofs can be divided into two categories: interactive and automatic. Interactive theorem provers (ITPs) allow users to define conjectures, then attempt to manually write proofs for them, relying on the tool to organize the set of hypothesis and how the proof obligations 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), their implementation usually just straightforwardly checks the logic rules from the logic framework that is being implemented. Because of this, each step must be explicitly stated by the user, making the tool costly to be used. However, some ITPs offer the possibility to facilitate the process of writing proofs through the usage of tactics. A tactic is a procedure that can potentially inspect the set of hypothesis and the proof obligations and manipulate them. Usually they are used to simulate common proof techniques, such as case analyses and induction. The validity of each tactic call is verified by the kernel of the ITP to ensure that it does not compromise the tool's soundness.

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 otherwise. There are a variety of tools that use different techniques to automatically reason about logical propositions. We will focus on Satisfiability Modulo Theories [12, ch. 33] (SMT) solvers, systems that determine the satisfiability of formulas written in an extended version of First-Order Logic, capable of incorporating structures from external domains such as arithmetic, arrays and others. SMT solvers employ a combination of a SAT solver and domain-specific methods to solve such formulas. 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, further developments of the ATP become harder, since the changed system has to be verified again.

Another approach to increase the confidence in ATPs is to have them provide a certificate to support the result it found for a particular conjecture. A certificate is a counter-example if the conjecture was false or a mechanized proof if it was true. One can then independently verify whether this certificate indeed proves the theorem in question. This approach has the downside of creating a need for checking the certificate for each theorem proved. On the other hand, as long as the proof format does not change, the implementation of the solver can be modified without requiring a modification in the checkers. Also, verifying the correctness of certificates is often much simpler than to verify the tool itself, or to solve the original problem.

Another important advantage of the second approach is that it allows the ITPs to leverage the automatic proving performed by the ATPs via the certificates 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 is possible for the user of the ITP to focus on more complex steps of the proof, while delegating the burden of other long but straightforward steps to the ATP. Indeed, there are many examples of projects that successfully implement this sort of connection between ATPs and ITPs, which will be presented in Section 1.2.

1.1 Contributions

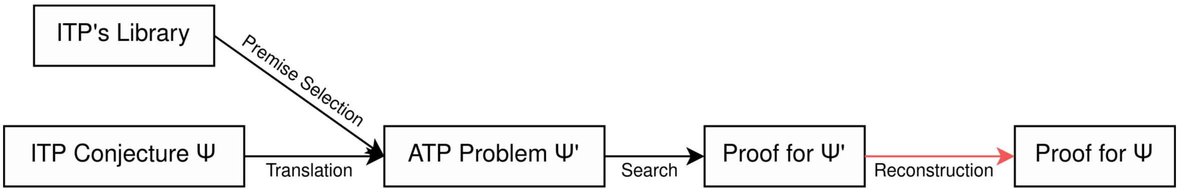
We have built a system that takes proofs of the unsatisfiability of SMT queries produced by the SMT solver cvc5 [3] and reconstructs them as proofs in the ITP Lean 4 [23], which has as its main goal the capacity of reconstructing the largest possible range of proofs produced by cvc5. We present in detail two strategies well-known in the literature for doing this translation. We have effectively applied one of them, and we share our insights regarding the reasons we believe it is more appropriate than the other for our goals.

The main motivation of this project is that, despite the fact that Lean is emerging as a promising programming language and ITP and being widely used by mathematicians in large-scale formalizations [18, 33], 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 Section 1.2. The contribution of the present work is essential to develop this kind of project using Lean.

The cvc5 solver has a module for post-processing its proofs, translating and printing them in a format recognized by Lean [4] (which is the reason why we chose it). Each proof produced by cvc5 consists of a set of logical inferences starting from the hypothesis until the goal is inferred. In order to reconstruct such proofs inside of Lean, it is necessary to prove the correctness of these inference rules inside the ITP. Our main contribution is to provide a set of tactics and theorems matching the set of rules used by the ATP, enabling the verification of the translated proofs through Lean's kernel. Furthermore, it is possible to leverage these proofs to discharge the responsibility of proving theorems originally stated in Lean to cvc5, which is the main use case for our module. There are other modules that need to be implemented in order to fully automate this process (which we will describe in Section 1.2), that are out of the scope of the present project. However, our project is being used as part of the joint project LeanSMT[1], that aims to implement a tactic in Lean that would perform this process of discharging proofs. Starting from a Lean goal, the tactic would translate it to an SMT query, then invoke the solver to try to prove it and lift 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] The code for the project can be found at https://github.com/ufmg-smite/lean-smt

Figure 1.1: General Architecture of a Hammer



Font: Elaborated by the author.

Another contribution of our work is the first checker for cvc5 proofs (which are expressed in the theories we support) verified by an established ITP. Other checkers exist ([1] and [44], for instance), but one has to trust their source code to verify proofs with them. While it is necessary to trust Lean's kernel to verify proofs with our module, Lean is a tool with thousands of users constantly writing proofs in it. Therefore, if it had an inconsistency in its kernel, it is very likely that an user would have already discovered it. While this is an useful feature of our tool, it is important to highlight that we did not design it as a proof checker and it lacks optimization for this purpose, resulting in a suboptimal performance for this task.

1.2 Related work

Systems whose purpose is to discharge the responsibility of proving theorems stated in an interactive theorem prover to an automatic one are known as “hammers” [13]. Given a conjecture ψ to be proven inside an ITP, a hammer will execute the following three steps:

- First, it will inspect the library of theorems already proven inside the ITP and heuristically select the ones that are more relevant to prove ψ. Usually, the ATP will not be aware of the existence of these theorems, so it is possible to facilitate the process of searching for a proof by adding them as hypothesis to the problem. The module responsible for this process is known as the **premise selection module**.

- Once the premisses are selected, the hammer will translate ψ into a problem ψ' expressed in a language recognized by the ATP. The problem ψ' must be equivalent to ψ, with all the premisses that were selected in the previous step added as hypothesis. This is done by the **translation module**.

- Next, ψ' will be fed to the automatic theorem prover. The solver will then try to verify the correctness of the conjecture or find a counterexample, producing a certificate supporting the result it found. If the ATP could validate the conjecture, then the certificate will be a proof. In this case, the hammer will use the **proof reconstruction module** to reconstruct this proof inside the interactive theorem prover as a proof for ψ.

This workflow is summarized in Figure 1.1.

In this dissertation we describe how we implemented a proof reconstruction module for Lean users from cvc5 proofs. The two main strategies used to reconstruct the proof produced by the automatic system inside the interactive are also well-known: The first one is known as the *certified* approach [16]. In this case, the hammer defines a datatype to represent terms in the language of the ATP and a set of functions to manipulate values of those datatypes, representing the inference rules that the solver uses to reason about those terms. Then, a lifting function is defined, that is, a function that takes a value of this datatype and outputs an equivalent term in the native language of the ITP. Finally, the correctness of each function is verified with respect to the lifting function, in the sense that, if the input term was lifted to a value that is true in the ITP's logic, then the output term will also be true. The ATP's proof will be represented as a sequence of applications of those functions, and their correctness are proved *a priori*. When checking a specific proof, the only step that the hammer 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.

The second approach is known as the *certifying* approach [16]. It consists of matching each axiom in the ATP's logic into tactics defined in the ITP that operate directly over native terms of the system and parsing the proof produced by the automatic solver into a sequence of applications of those tactics, which are replayed inside the ITP. Those tactics rely on theorems previously proved inside the ITP to simulate the ATP's inference rules. In this case, the proof steps are reconstructed and checked in the ITP on the fly each time the hammer is invoked. Since there is no theorem stating the correctness of them, it is possible that this process fails. On the other hand, this technique skips computations over embedded terms which have to be done by the certified approach, granting it the potential to have a better performance [2]. It also diminishes the complexity of the tool, as there is no need to define an intermediate representation inside the ITP.

We present implementations of the certified and certifying approaches in Lean in Chapters 3 and 4, respectively. For reasons we will explain in the later chapters, our main focus is on the certifying approach.

The next two sections describe examples of hammers. Both of these works exhibit similarities to our research, therefore, it is useful to reference them here. It is important to note that both of these are established hammers, and teams have been working on optimizing their performance for years. As we have previously pointed out, the primary objective of our system is to cover the largest range of proofs produced by cvc5 as opposed to having the best possible performance. Therefore, due to the time constraints of this project, improving the efficiency of our module is left for future work, so the current system is slower than the other hammers. In Section 6, we discuss ideas to improve this aspect of our tool.

1.2.1 SMTCoq

SMTCoq [25] is a hammer for the Coq proof assistant [11]. It offers tactics to prove theorems in Coq via the external proof-producing SMT solvers veriT [17] and CVC4 [6]. The tool can support multiple proof formats due to a preprocessor written in OCaml, that is able to turn them into a unified certificate in the Coq language, which will be used as input for the plugin. Our tool explores one specific proof format of cvc5, therefore, for now, it only supports this solver and is less flexible in this aspect than SMTCoq.

The core of the hammer directly follows the certified transformations approach (although recently, it was extended with new features, implemented using certifying transformations [16]). It has a set of certified functions representing the transformations that are sound in three of the main domains that are available in SMT: Linear Arithmetic, Uninterpreted Functions and Bitvectors. The first one is used to represent formulas involving numeric variables, the second one is used to represent congruence relatons over variables and uninterpreted functions and the third one is used to simulate operations over numbers as they are represented by computers, preserving the semantics of this representation. Our tool currently supports Linear Arithmetic and Uninterpreted Functions.

The way that SMT solvers prove that a proposition is true is by showing that the negation of the proposition is unsatisfiable, that is, false for any assignment of its free variables. This kind of proof is parsed in SMTCoq into a sequence of applications of certified functions, which have to transform the term representing 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 hammer apply its main theorem, which states that if this process was successful, then the original goal is true.

SMTCoq also differs from our work as it implements a translation module, but it does not implement a premise selection module.

1.2.2 Sledgehammer

The ITP Isabelle/HOL [37] has a similar tool, called, Sledgehammer [14]. This system has multiple strategies for building a proof for a theorem using ATPs. One of them is to invoke several ATPs in parallel on the given goal and parse their output into a sequence of applications of Isabelle's predefined lemmas and tactics that could possibly prove the original goal. This falls into the second category of strategies for proof reconstruction presented in Section 1.2.

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 premise selector (as described in Section 1.2) for the other plugins.

1.3 Organization of this document

In Chapter 2 we present the concepts involved in our work. We describe the SMT problem in detail, as well as the main techniques used to solve it and the proof certificates produced by cvc5. We also present Lean and give an overview of the features of the language we have employed throughout the project. Then, in Chapter 3, we describe the steps necessary to implement the reconstruction of proofs from cvc5 in Lean using the approach of certified transformations. Next, in Chapter 4, we present in detail our implementation of the reconstruction using the certifying transformations approach. This implementation is evaluated in Chapter 5. Finally, we present some possible directions for future work in Chapter 6.

Chapter 2

Formal Preliminaries

2.1 Satisfiability Modulo Theories (SMT)

In this chapter we present the SMT problem. Recall that our goal is to translate proofs of unsatisfiability of SMT queries into Lean proofs. Therefore, besides presenting formally the problem and explaining how it is solved, we focus on showing how proofs are represented in an SMT solver.

2.1.1 Description of the Problem

The Boolean Satisfiability Problem (SAT) consists of determining whether a formula in Propositional Logic, which contains free variables, can be evaluated to *true* by finding a function that assigns a Boolean value to each variable. We say that a formula is satisfiable if such a 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 Conjunctive Normal Form, that is, a conjunction of disjunctions of literals. From now on, we will use SAT to refer to the CNF-SAT problem. Furthermore, we will write *clause* to refer to each one of the disjunctions in some input formula for SAT. We will also treat a clause as a set of its composing literals (a literal is either a variable or the negation of a variable) and use set operations over it. For instance, we will write $x \in C$ to state that the literal x is present in clause C and $C \setminus \{x\}$ to refer to the clause C without the literal x. Also, we extend this abstraction to include the clause corresponding to the empty set, which can never be satisfied.

Satisfiability Modulo Theories (SMT) [12, ch. 33] is a generalization of SAT. There are two additions: first, the underlying logic framework is Equational First-Order Logic instead of Propositional Logic. This means that the input formula can contain quantifiers binding variables, unintepreted function symbols, predicates asserting properties about variables and a special symbol "$\simeq$" that is used to assert equality between terms. The second addition is the inclusion of a set of sorts, equipped with predefined operations that allow the problem to refer to different domains. For example, one instance of a problem in this extension can refer to variables ranging over integers and contain assertions that compare the values of such variables. This extension also permits the same instance of the problem to refer to multiple distinct domains. The logic framework that corresponds to Propositional Logic with these two additions is known as Many-Sorted First Order Logic (MSFOL). The precise syntax and semantics of this logic framework is given in detail for example in [32]. In Section 2.1.2 we give a brief overview of it.

2.1.2 Many-Sorted First Order Logic

Syntax

The syntax of MSFOL consists of three syntatic categories:

1. **Sorts**. Symbols identifying the kinds of variables that are allowed. They are defined by the following grammar:

   $$\tau ::= \sigma \mid (\tau_1, \tau_2, \ldots, \tau_n) \rightarrow \tau$$

   The first case represents an atomic sort drawn from a predefined set of sorts, which we will refer to as $\mathcal{S}_S$, and is required to have a cardinality that is at most countably infinite. The second case represents sorts of functions with arity n whose return type is τ. We will always assume that there is a special sort in $\mathcal{S}_S$ named *bool.*

2. **Terms**. Symbols representing sorted variables, constants or function applications. Function applications are only well formed when the sorts are respected. The symbols annotated on top of each term represent their sort, and they will be ommited when they can be inferred.

   $$t ::= x^\tau \mid f^{(\tau_1,\ldots,\tau_n)\rightarrow\tau}(t_1, \ldots, t_n)$$

   where the return type of f must be different than *bool*. Function symbols are also drawn from a predefined set, which we will refer to as $\mathcal{S}_F$, and it is also required to have a cardinality that is at most countably infinite. Constants are represented by nullary functions. Variables will also be drawn from a predefined set, which we will refer to as $\mathcal{S}_X$ and, again, impose the requirement of having a cardinality that is at most countably infinite.

3. **Formulas**. Expressions built with logical connectives representing statements.

$$\psi ::= p^{(\tau_1,\dots,\tau_n)\to bool}(t_1,\dots,t_n) \mid \psi_1 \vee \psi_2 \mid \forall x^\tau.\psi \mid \neg\psi \mid \psi_1 \simeq \psi_2$$

The first case is an *atom*, that is, an application of a function whose return type is *bool*. This kind of function will be referred to as *predicate*.

We also define the following symbols:

$$\begin{aligned}\exists x.\psi &:= \neg\forall x.\neg\psi\\ \psi_1 \wedge \psi_2 &:= \neg(\neg\psi_1 \vee \neg\psi_2)\\ \psi_1 \to \psi_2 &:= \neg\psi_1 \vee \psi_2\\ \psi_1 \leftrightarrow \psi_2 &:= \psi_1 \to \psi_2 \wedge \psi_2 \to \psi_1\\ \psi_1 \not\simeq \psi_2 &:= \neg(\psi_1 \simeq \psi_2)\end{aligned}$$

We define a *signature* to be a triple of sets $\langle \mathcal{S}_S, \mathcal{S}_F, \mathcal{S}_X\rangle$ respecting the requirements defined above.

Semantics

Having established how terms in MSFOL are built, we have to define how they are evaluated, i.e. their meaning. Given a signature $\Sigma = \langle \mathcal{S}_S, \mathcal{S}_F, \mathcal{S}_X\rangle$, a Σ-*structure* is a function $\mathcal{I}$ that maps, for each sort symbol $s \in \mathcal{S}_S$, a corresponding set $\mathcal{I}(s)$, for each function symbol $f^{(\tau_1,\dots,\tau_n)\to\tau} \in \mathcal{S}_F$, a function $\mathcal{I}(f)$ of type $\mathcal{I}(\tau_1) \times \dots \times \mathcal{I}(\tau_n) \to \mathcal{I}(\tau)$ and for each variable $x \in \mathcal{S}_X$ of sort s, a value $\mathcal{I}(x) \in \mathcal{I}(s)$. The *bool* sort will always be mapped to the usual set of Boolean values, and the set $\mathcal{S}_F$ will always contain two nullary functions, *true* and *false*, which will be mapped to the usual true and false values, represented here respectively as $true^I$ and $false^I$. A Σ-*Theory* is a set of Σ-structures. Theories are used to refer to the multiple possible assignments of values to variables in a formula.

Finally, we define the *evaluation* of a given formula ψ over a Σ-structure $\mathcal{I}$, denoted by $[\![\psi]\!]^{\mathcal{I}}$, in the following way:

$$\begin{aligned}[\![x]\!]^{\mathcal{I}} &:= \mathcal{I}(x)\\ [\![f(t_1,\dots,t_n)]\!]^{\mathcal{I}} &:= \mathcal{I}(f)([\![t_1]\!]^{\mathcal{I}},\dots,[\![t_n]\!]^{\mathcal{I}})\\ [\![\bot]\!] &:= false^I\\ [\![t_1 \simeq t_2]\!]^{\mathcal{I}} &:= [\![t_1]\!]^{\mathcal{I}} = [\![t_2]\!]^{\mathcal{I}}\\ [\![\neg\psi]\!]^{\mathcal{I}} &:= [\![\psi]\!]^{\mathcal{I}} = false^I\\ [\![\psi_1 \vee \psi_2]\!]^{\mathcal{I}} &:= [\![\psi_1]\!]^{\mathcal{I}} = true^I \textit{ or } [\![\psi_2]\!]^{\mathcal{I}} = true^I\\ [\![\forall x^\tau.\psi]\!]^{\mathcal{I}} &:= [\![\psi]\!]^{\mathcal{I}_{x\leftarrow v}} = true^I, \text{ for any } v \in \mathcal{I}(\tau)\end{aligned}$$

where $\mathcal{I}_{x \leftarrow v}$ denotes the extension of the function $\mathcal{I}$ in which the variable x is assigned the value v. We say that an structure $\mathcal{I}$ *satisfies* a formula ψ if and only if $[\![\psi]\!]^{\mathcal{I}} = true^{I}$. The structures that satisfy a formula ψ are known as the *models* for ψ. If there is at least one model for a given formula ψ, it is said to be *satisfiable*, otherwise it is *unsatisfiable*.

Example 1 (LIA)**.** *Let $\mathcal{S}_S := \{Z\}$, $\mathcal{S}_F := \{add, sub, zero, succ, pred, lt\}$ and $S_X := \{x, y\}$. Examples of formulas that can be written using the signature composed by these sets include "$lt(sub(x, y), succ(zero))$" and "$add(x, y) \simeq add(y, x)$". For this particular signature, we can have many theories. For instance, if we fix $\mathcal{I}(Z) := \mathbb{Z}$ and the interpretation of each function symbol to its usual operation over integers, we can have an infinite set of structures, each one mapping x and y to a pair of integers. This set is a theory known as Linear Integer Arithmetic (LIA), which is able to express formulas involving linear arithmetic expressions over integers. On the other hand, if we let $\mathcal{I}(Z) := \mathbb{Z}_{13}$ and match each function symbol with the corresponding operation over integers modulo* 13*, we have a theory representing formulas over integers modulo* 13*.*

2.1.3 SMT Solvers

An SMT solver is a piece of software whose main goal is to solve the SMT problem, that is, determining whether a given formula expressed in MSFOL is satisfiable or not. This problem is undecidable in its most general form. However, some variants of it which restrict the input formula to be expressed over certain theories (such as LIA) are known to be decidable. Thus, in order to achieve their goal, SMT solvers utilize a combination of heuristics (for managing undecidable theories) along with decision procedures (for those theories known to be decidable). In this section we present the methods employed by these systems that are more relevant to the present work.

Davis–Putnam–Logemann–Loveland Algorithm (DPLL)

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

A more efficient alternative of a decision procedure for PL is the DPLL algorithm [20]. DPLL is based on the *Resolution* theorem:

Figure 2.1: DPLL Algorithm

```
1: function SolvePL($\psi$)
2:     if $\exists C \in \psi . C = \{\} \vee C = \{\bot\}$ then
3:         return $false$
4:     end if
5:     if $\forall C \in \psi . \top \in C$ then
6:         return $true$
7:     end if
8:     if $\exists x \in Vars(\psi)$ such that $x$ is a target for UR then
9:         $\langle C_1, C_2 \rangle \leftarrow$ findClauses($x$, $\psi$)          ▷ Suitable for applying UR with $x$
10:         return SolvePL($\psi \wedge (C_1 \setminus \{x\} \vee C_2 \setminus \{\neg x\})$)
11:     end if
12:     Let $x$ be an unassinged variable in $\psi$
13:     return SolvePL($\psi_{\{x \leftarrow \top\}}$) $\vee$ SolvePL($\psi_{\{x \leftarrow \bot\}}$)
14: end function
```

Font: Elaborated by the author.

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

More specifically, it is based on *Unit Resolution* (UR), that is, a more restricted version of resolution in which $C_1 = \{x\}$ or $C_2 = \{\neg x\}$. Notice that the clause that is inferred from the unit resolution theorem is equal to one of the clauses in the premisses, with one less literal. The general idea of the DPLL algorithm is to explore this fact to generate smaller clauses, while this is possible. Once all possible unit resolutions are performed, the algorithm makes a decision, assigning a boolean value to a variable. This action can potentially create more clauses suitable for unit resolution, which will be then explored. If this decision does not lead to the conclusion that the original formula is satisfiable, the procedure backtracks and assigns the other boolean value to the variable. If this decision also does not lead to a positive conclusion, then the original formula is unsatisfiable.

Figure 2.1 shows an algorithm for DPLL. The procedure *SolvePL* works as follows: first, in lines 2 to 7, it checks if the formula can be evaluated to *true* or *false*. In case it is not possible, the procedure finds as many variables in which it can apply Unit Resolution as possible, using the routine *findClauses*, and invokes itself recursively with each new application found, as shown in lines 8 to 10. 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 as input is also satisfiable, therefore, this step is sound. Once there are no more possibilities, it chooses an arbitrary variable and makes two recursive calls in line 13: 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.

For instance, let's run the algorithm with $\psi := \{\neg x, x \vee y, x \vee \neg y\}$. Neither of the checks of lines 2 and 5 will succed with this formula. The check on line 8 will, since we can apply unit resolution on the first and second clauses, eliminating the variable x. This will produce the new clause y. Following the algorithm, we apply the function $solvePL$ using $\{\neg x, x \vee y, x \vee \neg y, y\}$ as the argument. The checks on lines 2 and 5 will fail again. We can apply unit resolution again using the clauses $\neg x$ and $x \vee \neg y$, eliminating x. This will produce the clause $\neg y$. We add it to the input formula and run the function again. Once more, the checks on lines 2 and 5 will fail, and we will be able to apply unit resolution to the clauses y and $\neg y$, resulting in the empty clause. Now, when we invoke the function recursively again, the check on line 2 will succed, and it will correctly return $false$.

The actual algorithm used by most SMT solvers to solve SAT is a refinement over DPLL, called Conflict Driven Clause Learning (CDCL) [21]. Its main idea is to modify the DPLL algorithm so that, when it finds a conflict (an empty clause or a clause that only contains false variables), it analyzes the reason for this conflict, allowing it to derive new clauses and to backtrack many decisions of values of variables at once. The DPLL algorithm also obtain information from a conflict, but it cannot backtrack many decisions at once. The new clauses derived from the analysis performed by CDCL can also be inferred via resolution. Since our main concern in this work is on proofs, and, as we will show, proofs for unsatisfiability of Boolean formulas are given in terms of resolution inferences, this refinement is not relevant for our work and we will not present it in detail here.

Proof Certificates for Boolean formulas

In order to convince the ITP that the result found by the SMT solver is correct, we need evidence provided by the ATP. We will refer to such evidence as proof certificate. The main feature of a good certificate is being easy to check. In particular, it is desirable that it is easier to check a certificate then to solve the original problem. The theories that are of our interest in this project have already been instrumented in cvc5 enabling the solver to produce proof certificates for them [4].

If the SMT solver found out that a given formula is satisfiable, then it can always provide the model for that formula as a certificate. If the input formula is unsatisfiable, then this option is not available, so it is more challenging to provide such certificate. In this work we will focus on proof certificates for the unsatisfiability of formulas.

The main proof certificate produced by SMT solvers for the unsatisfiability of instances of the SAT problem is a *resolution tree*. A resolution tree is a binary tree in which each leaf correspond to a clause from the SAT instance, each internal node correspond to the clause resulting of applying the Theorem 1 on its two children and the root corresponds to the empty clause.

Figure 2.2: CDCL(T) Algorithm

```
function SolveMSFOL($\psi$)
    $\psi_{CNF} \leftarrow$ ConvertCNF($\psi$)
    $\psi_{PL} \leftarrow$ Convert($\psi_{CNF}$)            ▷ Get PL formula from MSFOL formula
    if CDCL($\psi_{PL}$) then
        $\mathcal{I} \leftarrow$ GetModel($\psi_{PL}$)
        if TheorySolver($\mathcal{I}, \psi$) then            ▷ Check if $\mathcal{I}$ is compatible with $\psi$
            return true
        end if
        $L \leftarrow$ GetConflictingLemma($\mathcal{I}, \psi$)
        return SolveMSFOL($\psi \wedge L$)
    end if
    return false
end function
```

Font: Elaborated by the author.

For instance, for the example we provided for the DPLL algorithm, the resolution tree that corresponds to that particular execution of the algorithm is the following:

$$\dfrac{\dfrac{\neg x \quad x \vee y}{y} \qquad \dfrac{\neg x \quad x \vee \neg y}{\neg y}}{\bot}$$

CDCL(T)

Assuming that we have a decision procedure for a given theory (or a combination of them), we can use it to extend the CDCL algorithm for checking the satisfiability of MSFOL formulas involving that theory. In this section we will study the CDCL(T) [36] framework, a method for doing this extension over CDCL.

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 conjunctive set of propositions in $\mathcal{T}$ is consistent or not). We do not need the theory solver to handle disjunctions between propositions since the CDCL(T) algorithm will leverage a SAT solver to handle this part of the analysis. The formula ψ might not be in conjunctive normal form. Therefore, the first step is to obtain a formula ψ_{CNF} that is in this format and is satisfiable if and only if ψ is satisfiable. This is usually done through the Tseitin transformation [45] to avoid an exponential grow in the size of the formula. The next step is to create a PL formula ψ_{PL} from ψ_{CNF} by substituting each atom built from a predicate of $\mathcal{T}$ in it for a fresh Boolean variable. We can then use the previously described CDCL algorithm to determine whether ψ_{PL} is satisfiable. 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 $x < -2$. If ψ_{PL} is unsatisfiable, then ψ is also unsatisfiable. If this was not the case, it would be possible for ψ_{PL} to be unsatisfiable and ψ be satisfiable. This means that there would exist a model for ψ. From this model, we would be able to derive truth values for the variables in ψ_{PL} that should satisfy the formula, but this is an absurd since ψ_{PL} is unsatisfiable. If ψ_{PL} is satisfiable, we can find a model $\mathcal{I}$ for ψ_{PL}. Although $\mathcal{I}$ satisfies ψ_{PL}, it is possible that it is contradictory in the context of the theory $\mathcal{T}$. In our previous example, the only possible model for ψ_{PL} is the one that assigns p and q to $true^I$, but this is not valid when we translate back to ψ, as x cannot be both greater than 3 and smaller then -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 $\neg(x > 3 \wedge x < -2)$. Note that we are assuming that our theory solver has the capability of producing such a conflicting lemma for a particular model.

Figure 2.2 shows an implementation of CDCL(T). The procedure *SolveMSFOL* works as follows: first, in line 2, we convert the formula to CNF. Then, in line 3 perform the conversion that we just described, in order to obtain a PL formula. Next, in line 3, we run the procedure *CDCL* to determine if ψ_{PL} is satisfiable. In case it is not, we just return *false* in line 12. Otherwise, there exists an model $\mathcal{I}$ that satisfies ψ_{PL}. This model is obtained in line 5 and we proceed to check if the theory solver accepts it for ψ in line 6. If it does accept, the formula is definitely satisfiable and we can return *true*, as done in line 7. Otherwise, the conflicting lemma for the given model is generated in line 9. We then repeat this process in line 10, adding the conflicting lemma to ψ.

Proofs produced by the CDCL(T) algorithm will be proof trees that combine resolution steps with lemmas that are particular to each theory. Those lemmas will be discussed in the next sections. Furthermore, the process of transforming the input formula into CNF also has to be justified in the proof certificate. Therefore, the proof trees produced might also contain steps justifying this transformation. These steps usually are simple rules from Boolean reasoning, such as the De Morgan's laws.

Equality and Uninterpreted Functions

In this work, we will be interested in two theories and their respective solvers. The first one is Equality and Uninterpreted Functions.

Consider the following problem: given a set of variables and uninterpreted functions and a set of equalities built from variables and functions applied to those variables, decide whether a given equality is a consequence of the conjunction of the ones in the set. For instance, let a, b, c and d be variables and f a function of arity 1. Assume that all variables belong to an artificial sort U, and all functions both take arguments and return values of this sort. The set $S := \{b \simeq c, f(b) \simeq c, f(c) \simeq a\}$ has as one consequence the equality $a \simeq b$.

Given a set S of equalities between terms, the set of all equalities (only involving those terms) that can be derived from the elements of S is known as the *congruence closure* of S. Deciding whether a given equality is in the congruence closure for a given

Figure 2.3: Merge with Congruence

```
procedure MergeCong($v_1$, $v_2$)
    if $\neg$ SameClasses($v_1$, $v_2$) then
        Let $v_1^*$ and $v_2^*$ be the equivalence classes of $v_1$ and $v_2$.
        Let $P_{v_1^*}$ be the set of all predecessors of all vertices in $v_1^*$
        Let $P_{v_2^*}$ be the set of all predecessors of all vertices in $v_2^*$
        Merge($v_1$, $v_2$)
        for each $u_1 \in P_{v_1^*}$ and $u_2 \in P_{v_2^*}$ do
            if IsCongruent($u_1$, $u_2$) then
                MergeCong($u_1$, $u_2$)
            end if
        end for
    end if
end procedure
```

Font: Elaborated by the author.

set is an old problem, which was shown to be decidable by Ackermann in 1954 [9]. It is also is used as the basis for the theory of Equality and Uninterpreted Functions (EUF) in SMT, as we will see later.

This theory is used to express satisfiability problems involving equalities and inequalities of variables and function applications. Since the language of MSFOL already has a built-in equality operator, we do not need to fix the meaning of any operator in this theory. Its semantics are the same of Equational First-Order Logic without any addition. For this reason, this theory is also known as the “empty theory”.

We now give a brief overview on the standard algorithm for finding the congruence closure of a given set of equalities. A more detailed explanation can be found at [35]. This solution is based on the concept of congruence. Congruence is a property that all function applications enjoy: if f is a function of arity n and $(a_k)_{k=1}^n$ and $(b_k)_{k=1}^n$ are two sequences of terms such that $a_i \simeq b_i$ for all i, then $f(a_1, \ldots, a_n) \simeq f(b_1, \ldots, b_n)$.

Let S be the set of equalities that are assumed and e be the one that we want to decide whether is a consequence of S. Let T be the set of all terms that appear in $S \cup \{e\}$ including all function parameters. Let's build a graph G in which for all $t \in T$ we will have a vertex v in G that represents that term. For the vertex v, we will assign a label with the name of the term it represents. Moreover, if t is a function application we will add directed edges from t to each vertex that represents a term that is an argument to t. Also, we will keep an order over the children for each vertex, according to the order in which the parameters are applied. For instance, let $S := \{f(a, b) \simeq a\}$ and $e := f(f(a, b), b) \simeq a$. Then we would build G as follows:

For a vertex v, we will denote by $\lambda(v)$ its label, $\delta(v)$ its outdegree and $Child(v, i)$ its i-th child. Next, we will keep a data structure that is capable of representing a set of disjoint sets of vertices. Each one of these disjoint sets represents a class in an

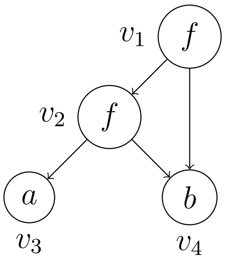


Figure 2.4: Check Congruence Condition

```
function IsCongruent($v_1$, $v_2$)
    if $\lambda(v_1) \neq \lambda(v_2)$ or $\delta(v_1) \neq \delta(v_2)$ then
        return $false$
    end if
    for $i = 1$ to $\delta(v_1)$ do
        if $\neg$ SameClasses(Child($v_1$, $i$), Child($v_2$, $i$)) then
            return $false$
        end if
    end for
    return $true$
end function
```

Font: Elaborated by the author.

equivalence relation, where every vertex in the same class is known to be equal. This data structure must also be able to check whether two vertices are on the same class and to join two classes. An example of structure with such capabilities that is efficient and easy to implement is the Union-Find [26].

Initially, each term is only equal to itself. Then, for each equality $t_1 \simeq t_2$ in S we will use the procedure *MergeCong* presented in Figure 2.3 to merge the vertices v_1 and v_2 that correspond to t_1 and t_2. Notice that, when we process an equality $t_1 \simeq t_2$, we also have to propagate this information to any other terms that are using t_1 and t_2 as parameters (e.g. if we have $f(t_1)$ and $f(t_2)$ as terms, we must also merge their classes). That's why we recursively call *MergeCong* on line 11. This method has an auxiliary procedure for checking whether two vertices are congruent given the current state. Figure 2.4 shows an implementation of this method, which essentially just checks the premise of the congruence property.

Once this process is done, we can check whether any pair of terms t_1 and t_2 is currently equal to each other by checking whether their corresponding vertices v_1 and v_2 are on the same equivalence class (i.e. checking whether $Find(v_1) = Find(v_2)$). Therefore, we can use this fact to solve our original problem of checking if the equality e follows from the conjunction of the equalities in S.

Given a conjunctive set of propositions in the theory of Equality and Uninterpreted Functions, we can check its satisfiability in the following manner: we build the graph mentioned earlier using all terms appearing in the set (both in equalities and in inequalities). Then, for each equality $t_1 \simeq t_2$ we call the procedure $MergeCong$ to merge the vertices corresponding to v_1 and v_2. Lastly, for each inequality $t_1 \not\simeq t_2$, we check if the vertices corresponding to t_1 and t_2 are on the same equivalence class. If this is the case for any inequality, the set is unsatisfiable. Otherwise, it is satisfiable.

Proof Certificates for EUF

The algorithm we presented for computing the satisfiability of a conjunctive set of atoms from EUF can be instrumented in a way that, if it derives that a formula is unsatisfiable, it produce a proof tree, similar to the resolution tree, that certificates this result. The only difference between this proof tree and the resolution tree is that, instead of using the resolution theorem to infer new propositions, the proof tree will use the axioms that equality satisfies (reflexivity, symmetry and transitivity), as well as the congruence property. For instance, consider the following set of propositions: $\{a \simeq f(f(a)), a \simeq f(f(f(a))), f(a) \not\simeq a\}$. One possible proof tree that certificates its unsatisfiability is the following:

$$\dfrac{\dfrac{\dfrac{a \simeq f(f(a))}{f(a) \simeq f(f(f(a)))}\,\mathit{cong} \quad \dfrac{a \simeq f(f(f(a)))}{f(f(f(a))) \simeq a}\,\mathit{symm}}{f(a) \simeq a}\,\mathit{trans} \qquad f(a) \not\simeq a}{\bot}$$

Linear Arithmetic

The second theory that we will be interested in this work is the theory of Linear Arithmetic. A term in this theory is an expression of the form:

$$\sum_{i=1}^{n} a_i x_i \bowtie b$$

Where each a_i and b are constants ranging over the real numbers, each x_i is a variable and $\bowtie$ is one of $\leq$, $<$ or $\simeq$ (we model $\geq$ and $>$ as the negation of $<$ and $\leq$). If we allow variables to range only over integers (represented by the *Int* sort), we have the Linear Integer Arithmetic theory, which was described in terms of MSFOL in Example 1. If variables can range over the real numbers (represented by the *Real* sort), the corresponding theory is known as Linear Real Arithmetic (LRA), which can be described using MSFOL a similar way.

We will now review the method presented in [24] for checking whether a given set of atoms from LRA is satisfiable. This is the method generally employed by SMT solvers (in particular, by cvc5) to solve problems in this theory. It is broadly employed due to its amenability to run incrementally, an important feature in the CDCL(T) framework. The

Figure 2.5: Change value of non-basic variable and update all basic variables

```
procedure Update($v_i$, $val$)
    for $v_j \in \mathcal{B}$ do
        $\beta(v_j) \leftarrow \beta(v_j) + a_{ji}(v - \beta(v_i))$
    end for
    $\beta(v_i) \leftarrow val$
end procedure
```

Font: Elaborated by the author.

paper also presents an extension of this method for LIA, which we will not show here for brevity.

The first step is to, for each atom $t_i \bowtie b_i \in \phi$ in which t_i is a sum of at least two variables, create a fresh variable s_i. Then, we will define two new sets of atoms: ϕ_A, such that its atoms are all of the form $s_i \simeq t_i$, that is, they state the equality between the new variables and the corresponding terms, and ϕ', that has all the atoms in ϕ but with each term t_i substituted by the corresponding s_i. For instance, let the set of atoms be $\phi := \{x - y \geq 3, x - 2y < 6, x - y \leq 10, x \geq 5\}$. From this, we would define $\phi_A := \{s_1 = x - y, s_2 = x - 2y\}$ and $\phi' := \{s_1 \geq 3, s_2 < 6, s_1 \leq 10, x \geq 5\}$. Note that ϕ and $\phi_A \wedge \phi'$ are equisatisfiable, since we can rewrite the equations from ϕ_A in ϕ' to obtain the atoms in ϕ.

Next we will represent the restrictions in ϕ_A as $Ax = 0$, where A is a matrix and x is a vector with all the variables in the problem (including the ones we added). We build the i-th row in A with the coefficients for each variable in the equation for s_i in ϕ_A. We will refer to the entry in the i-th row and j-th column of A as a_{ij}. If the original problem had n variables and we added m fresh ones, the matrix A will have m rows and $m + n$ columns. In our example, we would have the following:

$$A := \begin{bmatrix} 1 & -1 & -1 & 0 \\ 1 & -2 & 0 & -1 \end{bmatrix} x := \begin{bmatrix} x & y & s_1 & s_2 \end{bmatrix}^T$$

Now, our problem is reduced to finding a vector x that satisfies $Ax = 0$ and respects all restrictions in ϕ', which are all of the form $v_i \bowtie b_i$, where v_i is one of the variables and b_i is a constant. We will assume that there is no strict inequalities in ϕ' for brevity. For an extension of the method presented here for strict inequalities, consult [24]. Notice that in this case we can represent the restrictions in ϕ' as two sequences of coefficients l_i and r_i, representing a sequence of restrictions of the form $l_i \leq v_i \leq r_i$, as long as we allow l_i to be $-\infty$ in case there is no lower restriction on v_i and r_i to be $+\infty$ in case there is no upper restriction on v_i.

We will solve this problem by processing each restriction in ϕ' incrementally. The algorithm maintains a state that includes the coefficients l_i and r_i, as well as a function $\beta(v)$ that assigns a rational value to each variable v. At the beginning, since we haven't

Figure 2.6: Assert inequalities

```
1: function AssertLower($v_i$, $b_i$)
2:     if $b_i \leq l_i$ then return $true$ end if
3:     if $b_i > u_i$ then return $false$ end if
4:     $l_i \leftarrow b_i$
5:     if $v_i$ is a non-basic variable and $\beta(v_i) > u_i$ then
6:         Update($v_i$, $u_i$)
7:     end if
8:     return $true$
9: end function
1: function AssertUpper($v_i$, $b_i$)
2:     if $b_i \geq u_i$ then return $true$ end if
3:     if $b_i < l_i$ then return $false$ end if
4:     $u_i \leftarrow b_i$
5:     if $v_i$ is a non-basic variable and $\beta(v_i) < l_i$ then
6:         Update($v_i$, $l_i$)
7:     end if
8:     return $true$
9: end function
```

Font: Elaborated by the author.

processed any inequality yet, we will set each l_i to $-\infty$, each r_i to $+\infty$, and $\beta(v_i)$ to 0 for all v_i. Additionally, we will keep track of two sets of variables: $\mathcal{N}$ (non-basic variables) and $\mathcal{B}$ (basic variables). Initially, $\mathcal{B}$ will consist of all the variables we added to the problem (each s_i), while $\mathcal{N}$ will include all the original variables. Those sets can exchange elements during the execution of the algorithm, but we will always maintain the invariant that all non-basic variables respect their restrictions, that is, $l_i \leq \beta(v_i) \leq r_i$ for all $v_i \in \mathcal{N}$. Also, everytime we update the value of a variable we will also update the values of all basic variables, in order to satisfy the equation $Ax = 0$.

For each restriction in ϕ' of the form $v_i \leq b_i$ we will run the procedure *AssertUpper* shown in Figure 2.6, and for each restriction of the form $v_i \geq b_i$ we will run *AssertLower* (also shown in Figure 2.6). If there is a restriction of the form $v_i = b_i$ we will run both procedures.

Both functions return $false$ if and only if the current restriction is incompatible with the previous ones, otherwise they correctly update the bounds on the variable on the restriction, and also update its value in case it is a non-basic variable (remember that we keep the invariant that non-basic variables respect their bounds). If the current restriction is $v_i \leq b_i$, we first have to check if the current limit on v_i already satisfies this restriction, which is done in line 2 of the function *AssertUpper* and simply return *true* if this is the case. Otherwise, we compare the current lower bound on v_i with b_i and immediately return $false$ if it is not, since in this case it would be impossible to find a valid value for v_i. Finally, if none of these conditions held, we update the limit u_i

Figure 2.7: Update and pivot basic with non-basic variable

```
procedure PivotAndUpdate($v_i$, $v_j$, $val$)
    $\delta \leftarrow \frac{val - \beta(v_i)}{a_{ij}}$
    $\beta(v_i) \leftarrow val$
    $\beta(v_j) \leftarrow \beta(v_j) + \delta$
    for $v_k \in \mathcal{B} \setminus \{v_i\}$ do
        $\beta(v_k) \leftarrow \beta(v_k) + a_{kj}\delta$
    end for
    $\mathcal{B} \leftarrow (\mathcal{B} \setminus \{v_i\}) \cup \{v_j\}$
    $\mathcal{N} \leftarrow (\mathcal{N} \setminus \{v_j\}) \cup \{v_i\}$
end procedure
```

Font: Elaborated by the author.

accordingly in line 8 and update the value of v_i and of all basic variables in line 10, if it is necessary. The procedure *AssertLower* is symmetrical.

Lastly, if it was possible to run the assertion procedures on all the inequalities from the input without any of them returning *false*, we run the procedure *Check* presented in Figure 2.8. The goal of this function is to manipulate the values of the non-basic variables until all variables respect their restrictions, or report that it is impossible, and, therefore, the original formula is unsatisfiable. The first step it takes, in line 2 is to select a variable $v_i \in \mathcal{B}$ that is violating its bounds. If there is no such variable then all restrictions are satisfied, and it can just return *true*, as is done in line 4. If $\beta(v_i) < l_i$, then we must find a non-basic variable v_j such that we can modify it without violating its restrictions in a way that the value of v_i will increase. If a_{ij} is positive, increasing v_j will also increase v_i, therefore, in order to use v_j we need $\beta(v_j) < u_j$. Similarly, if a_{ij} is negative, we would need $\beta(v_j) > l_j$. If there is no suitable variable, then it's impossible increase the value of v_i, therefore the problem is unsatisfiable. This condition is checked on lines 9 and 16. Otherwise, we can find such v_j. In this case, we set the value of v_j to the appropriate limit and make v_i a non-basic variable and v_j a non-basic variable with the procedure *PivotAndUpdate* in lines 12 and 19. After this, we repeat the process until all restrictions were satisfied. The case where $\beta(v_i) > u_i$ is symmetrical. In [24], the authors prove that this method always terminates and returns *true* if and only if the original problem was satisfiable.

Certificates for Linear Arithmetic

The proof certificates for unsatisfiability of problems on the theory of linear arithmetic rely on the following variant of Farkas' lemma [41]:

Theorem 2 (Farkas' Lemma). *Consider a sequence of m restrictions, where the i-th one has the form* $\sum_{j=1}^{n} a_{ij}x_j \bowtie_i b_i$ *and each* $\bowtie_i$ *is either* $\leq$ *or* $<$. *If there is no solution for the*

Figure 2.8: Check if it's possible to satisfy all restrictions

```
function CHECK()
    Let $v_i$ be the basic variable that violates a restriction with the smallest index.
    if There is no such $v_i$ then
        return $true$
    end if
    if $\beta(v_i) < l_i$ then
        Let $v_j$ be the non-basic variable with smallest index such that
          $(a_{ij} > 0 \wedge \beta(v_j) < u_j) \vee (a_{ij} < 0 \wedge \beta(v_j) > l_j)$
        if There is no such $v_j$ then
            return $false$
        end if
        PIVOTANDUPDATE($v_i$, $v_j$, $l_i$)
    else if $\beta(v_i) > u_i$ then
        Let $v_j$ be the non-basic variable with smallest index such that
          $(a_{ij} < 0 \wedge \beta(v_j) < u_j) \vee (a_{ij} > 0 \wedge \beta(v_j) > l_j)$
        if There is no such $v_j$ then
            return $false$
        end if
        PIVOTANDUPDATE($v_i$, $v_j$, $u_i$)
    end if
    CHECK()
end function
```

Font: Elaborated by the author.

conjunction of these restrictions, then there exists a sequence of positive coefficients s_i*, known as Farkas' coefficients, that satisfies* $\sum_{i=1}^{m} s_i (\sum_{j=1}^{n} a_{ij}) \geq 0$ *and* $\sum_{i=1}^{m} s_i b_i < 0$.

One can use a sequence of coefficients with these properties to certify that a given instance of the problem is unsatisfiable. Given the sequence s, we can multiply each one of the restrictions $\sum_{i=1}^{n} a_i x_i \bowtie_j b_j$ by s_j, obtaining $s_j \sum_{i=1}^{n} a_i x_i \bowtie_j s_j b_j$ (note that $\bowtie$ does not change, as s_j is positive). Adding all the restrictions we get $\sum_{j=1}^{m} s_j (\sum_{i=1}^{n} a_i x_i) \bowtie^* \sum_{j=1}^{m} s_j b_j$, where $\bowtie^*$ is $<$ if at least one $\bowtie_j$ was $<$ or $\leq$ otherwise. Given that $\sum_{i=1}^{m} (\sum_{j=1}^{n} a_{ij}) \geq 0$ and $x_i \geq 0$ for all i, the left-hand side evaluates to a value greater than or equal to 0. Also, by the Farkas' lemma, the right-hand side evaluates to a negative value, which is an absurd, since it would imply $0 < 0$. Therefore, in order to verify that a given sequence of coefficients indeed prove that the restrictions are unsatisfiable, it's sufficient to apply the rules stating that we can add inequalities and multiply them by constants in the way we did, and then analyze the final inequality to obtain a contradiction.

For instance, consider the following set of restrictions: $\{x_1 + x_2 \leq 1, x_1 - x_2 \leq -2\}$ subject to $x_1 \geq 0$ and $x_2 \geq 0$. Let $s_1 = 1$ and $s_2 = 1$. The sum of the restrictions multiplied by the corresponding coefficients will be $1(x_1 + x_2) + 1(x_1 - x_2) = 2x_1$ and the sum of the constants multiplied by the coefficients will be $1 + 1(-2) = -1$. Therefore, we can derive the restriction $2x_1 \leq -1$, which is an absurd since $x_1 \geq 0$.

Since it is always possible to find these coefficients for unsatisfiable instances of the problem, the SMT solver can provide them as a proof certificate for this theory. Indeed, the algorithm we presented can be instrumented with little overhead to compute such coefficients. This is the certificate produced by cvc5, and it is the one we will be interested in this thesis.

2.2 Lean

Lean is both an Interactive Theorem Prover and a programming language. It is based on the Calculus of Inductive Constructions (CIC) [38] and explores the well-known correspondence between types and propositions [29] to implement a system that is both a proof checker and a type checker. In order to obtain extra safety, Lean is capable of exporting proofs that were checked by the kernel in a low-level format, which can be verified by external checkers[1]. Some of these checkers are quite small and can have their source code checked by the skeptical user. This is the case for instance of *trepplein*[2], which has just over 2000 lines of code[3].

Lean also has an impressive mathematical library driven by its growing community, namely, *mathlib* [33]. This library comprises over a million lines of code, formalizing various domains of knowledge from mathematics and theoretical computer science.

2.2.1 Lean as a Programming Language

Lean has all the main features one can expect from a functional programming language. Its features include algebraic datatypes, pattern matching, polymorphism, typeclasses, side effect support using monads, and a robust macro system [34].

The script in Figure 2.9 is a valid Lean program, that defines a new type representing natural numbers, together with a function for adding them. The keyword `inductive`

[1] As documented in: `https://github.com/leanprover/lean3/blob/master/doc/export_format.md`

[2] `https://github.com/gebner/trepplein/tree/master`

[3] This information was obtained by inspecting in the day 10/23/2023 the sum of the lines of each file in the folder src/main/scala/trepplein of trepplein's repository.

Figure 2.9: Example of Lean code

```
1   inductive Natural where
2     | zero : Natural
3     | succ : Natural -> Natural
4
5   open Natural
6
7   def add (n m : Natural) : Natural :=
8     match n with
9     | zero    => m
10    | succ n' => succ (add n' m)
11
12  notation x " + " y => add x y
```

Font: Elaborated by the author.

is used to introduce a new type in line 1, which in this case will be named `Natural`. After the name, the user must use the keyword `where`, followed by its constructors and their types (lines 2 and 3). The constructors for the type Natural are `zero` (which does not take any parameter and returns a Natural) and `succ` (which takes a Natural and returns another one). After Lean's kernel parses this statement, it adds the declaration of `Natural` to the context (i.e. that set of all declarations visible to the user). This constructors are introduced in the context with the names `Natural.zero` and `Natural.succ`. In order to be able to write just "zero" and "succ" we use the command `open Natural` in line 5.

Lines 7 to 10 define the sum function. We define new functions using the keyword `def` followed by its name, the list of parameters and their types, the return type and the body of the function after the symbol ":=". In this case, we define the function by pattern matching on the first parameter (line 8). If it is `zero`, we just return the second parameter, as shown in line 9. If it is `succ` of some other value `n'`, we return the `succ` constructor applied to the result of a recursive call of `add`, using `n'` and `m` (line 10).

Lastly, we use the command `notation` to define a macro for using the add function with the usual infix "+" operator in line 12.

2.2.2 Lean as a Theorem Prover

In Lean, propositions are represented as elements of the built-in type `Prop`. Also, they are themselves types, which are inhabited by terms that represent proofs of those statements. For instance, the following Lean expression represents a proposition stating

Figure 2.10: Proof that the addition of any natural number to zero results in itself.

```
1  theorem add_zero : ∀ (n : Natural), (n + zero) = n :=
2    fun n =>
3      match n with
4      | zero    => rfl
5      | succ n' => congrArg succ (add_zero n')
```

Font: Elaborated by the author.

that, according to our previous definition of natural numbers and addition, the sum of any natural `n` and `zero` results in `n`:

```
∀ (n : Natural), (n + zero) = n
```

The representation of this predicate as a type is possible due to the fact that Lean supports dependent types, that is, types that depend on values of other types. The operator `=` in this expression is a type constructor that accepts two values of the same type.

Therefore, proving that this statement holds amounts to finding a term with this type. The snippet in Figure 2.10 shows the construction of such term. The structure of this term follows the same pattern as the one for defining functions, the only difference is the change of the keyword `def` to `theorem`. After the symbol ":=" we have essentially a constructive proof of the statement. First, in line 2, it introduces the variable binded by the ∀ symbol in the context, using `fun n`. Then, in lines 3 to 5, it uses pattern match on `n`. If `n` is `zero`, the statement is reduced to `zero + zero = zero`, which follows directly from the definition of `add`. The term `rfl` is a proof that any term is equal to itself. In this case, Lean can match its type with the required one. If `n` follows the pattern `succ n’`, then the required type is `succ n’ + zero = succ n’`. By the definition of `add`, the left-hand side evaluates to `succ (n’ + zero)`. Note that the term `add_zero n’` has type `n’ + zero = n’`, which is almost the required one, missing only the `succ` on both sides. This is solved by applying `congrArg succ` on `add_zero n’` (`congrArg` is a theorem in Lean's library corresponding to the congruence property, which we have shown in Section 2.1.3), which produces a term with the correct type.

We also present a proof that our addition function is commutative, together with another necessary lemma:

```
theorem add_succ : ∀ (n m : Natural),
    (n + succ m) = succ (n + m) := fun n m =>
  match n with
  | zero    => rfl
  | succ n' => congrArg succ (add_succ n' m)
```

Figure 2.11: Proof of commutativity of addition using tactics.

```
1  theorem add_comm' : ∀ (n m : Natural), (n + m) = (m + n) := by
2    intros n m
3    induction n with
4    | zero       => rw [add, add_zero]
5    | succ n' IH => rw [add, add_succ, IH]
```

Font: Elaborated by the author.

```
theorem add_comm : ∀ (n m : Natural), (n + m) = (m + n) := fun n m =>
  match n with
  | zero     => Eq.symm (add_zero m)
  | succ n' =>
    Eq.trans (congrArg succ (add_comm n' m)) (Eq.symm (add_succ m n'))
```

Note that, since we have to make explicit every single logic step, even simple proofs are not easy to write and read. Lean (and most ITPs) provide an alternative for this kind of proof: tactics. As previously explained, tactics are routines that simulate proof techniques. While we are building a proof term, Lean's kernel always keeps track of a context, containing all declarations in scope and what is the currently expected type for the term we are building (also known as the goal). Tactics operate by manipulating these two structures, without compromising the trusted kernel. In other words, any modification that is made by a tactic must be properly justified and will be checked by Lean's kernel, in the same way it checked that our proof was correct.

We present a new version of `add_comm` with this new approach in Figure 2.11. The keyword `by` is used to indicate that the term will not be built explicitly, but instead computed by a sequence of tactics, also known as a tactic block. Given a goal of the form `∀ (v :  t), P(v)`, the tactic `intros x` will change the goal to `P(x)` and introduce in the context a variable with name `x` and type `t`. It can also be used to introduce multiple variables at one. We use it in line 2, to introduce two naturals, `n` and `m`. Next, we use the `induction` tactic in lines 3 to 5. This is a very general tactic that can apply the induction principle on any inductive type. Since our goal is of the form `P(n, m)`, where `n` is a natural number, it will produce two new goals: `P(zero, m)` and `P(n’, m) → P(succ(n’), m)`. Each one of these goals is then completed with the `rw` tactic. Given a term that represents a proof of an equality $\mathtt{e}_1 \simeq \mathtt{e}_2$, the tactic `rw` rewrites all occurrences of $\mathtt{e}_1$ by $\mathtt{e}_2$ in the goal. With this, we can avoid the usage of transitivity, symmetry and congruence lemmas that were needed on the other version of this proof. The tactic `rw` is implicitly invoking these lemmas for us. Note that this tactic also accepts function names such as `add` as a parameter. In this case, it rewrites the definition of the function.

Classical vs. Intuitionistic

The logic framework underlying Lean is intuitionistic [46]. This means that any proof written in the ITP must be done in a constructive manner. The correspondence between types and propositions that serves as a foundation for Lean is originally established in terms of constructive logic, therefore, this restriction is imposed to facilitate the concretization of this correspondence by the ITP.

The logic of the SMT solvers, on the other hand, is classical. This means that it accepts theorems that are impossible to prove constructively, such as the law of the excluded middle (i.e. for any proposition `P`, either `P` or `Not P` holds). Indeed, all the proofs that we will translate from cvc5 to Lean are based on double negation elimination, as we will always prove that some formula `P` is true by proving that `Not P` is unsatisfiable, which is also an implication that is not provable constructively.

Fortunately, Lean provides a mechanism for defining new axioms. These axioms will be available for the user to prove new theorems, which were possibly not provable before, extending the logic framework. Obviously, this must be done with caution, as there is nothing to prevent the introduction of axioms that will make the logic inconsistent. The `Classical` module, which is part of Lean's prelude, uses this feature to add the axioms needed to prove any true statement in classical logic. We used this module extensively in the present work.

2.2.3 Metaprogramming in Lean

Lean has a metaprogramming framework that enable its users to implement new tactics. In our project, we heavily rely on this framework for reconstructing the SMT solver's logical rules, as demonstrated later, in Chapter 4. We will provide a concise overview of the tactic implementation process in Lean. For a more comprehensive guide, refer to [39].

Expr

In Lean, the development of tactics relies on metaprogramming principles. These routines are allowed to access and manipulate terms using the internal representation employed by the compiler, granting them flexibility. Therefore, to understand how tactics operate it is important to understand how the compiler abstracts the structure of programs.

Terms (both values and types) are internally represented through the built-in type `Expr`, which models their abstract syntax tree. The following code shows a simplified version of the declaration of this type, omitting some specific parts that are not important in the context of this work (we also omit such parts in the examples we give):

```
inductive Expr where
| bvar    : Nat -> Expr
| fvar    : FVarId -> Expr
| mvar    : MVarId -> Expr
| const   : Name -> Expr
| app     : Expr -> Expr -> Expr
| lam     : Name -> Expr -> Expr -> Expr
| forallE : Name -> Expr -> Expr -> Expr
```

This type can be understood as an extension of the abstract syntax tree of terms in Lambda Calculus [5], where the constructors `bvar`, `app` and `lam` correspond to the rules for building terms in Lambda Calculus. The language structures that match each one of these constructors are given as follows:

- **bvar** (bounded variable): variables bounded by a lambda or a quantifier, such as the second `n` in `∀ (n : Natural), (n + zero) = n`. The `Nat` value they take as parameter is a natural number corresponding to its De Bruijn index [22].

- **fvar** (free variable): variables that appear in an expression which are not bound by a binder. There must be a declaration in context assigning a value for them. The parameter of their constructor (`FVarId`) is an identifier for their declaration in the context. Unlike bound variables, it is necessary to have access to the context to derive their type. The tactic `intros` we described before transforms bound variables into free variables on the goal.

- **mvar** (metavariable): variables that have a type but were not assigned an expression corresponding to their value yet. Goals that were not closed yet, for instance, are represented as metavariables. The parameter of their constructor (`MVarId`) is also an identifier for them in the context.

- **const** (constant): A constant value, previously declared. The `Name` parameter in the constructor is the internal type that represents identifiers. The syntax `` `ident `` creates a value of type `Name` representing the identifier `ident`. For instance, in the `Natural` type we introduced, `zero` is represented as `` const `zero ``.

- **app** (function application): Usual function application, in the style of Lambda Calculus. `succ (succ zero)` is represented as `` app (const `succ) (app (const `succ) (const `zero)) ``.

- **lam** (lambda abstraction): Usual lambda abstraction. The parameters represent: the name of the bound variable (used for pretty printing the expression); the type of the bound variable; and the body of the lambda expression.

- **forallE** (dependent arrow type): Used to represent types of functions. It also can express functions whose return type depends on the value it is given. This kind of type is also known as dependent arrows. The forall operator used before is syntax sugar for a dependent arrow. For instance, `∀ (n : Natural), (n + zero) = n` is the same as `(n : Natural) -> n + zero = n`. The first parameter of the constructor is the name of the variable in the type of the domain of the function (`n` in our example), the second is its type (`Natural` in our example) and the third is the return type of the function (`n + zero = n`) in our example. The return type can refer to the value in the domain type as `bvar 0`.

The end goal of a tactic block is to produce an expression that matches the type required by the theorem. Hence, the process of developing tactics is fundamentally based on manipulating expressions.

After all metavariables of an expression are assigned, the expression is sent to the kernel to be checked. Therefore, any manipulation done by a tactic will be checked and has to be sound, which means that we are not increasing the trusted base by using tactics.

MetaM and TacticM

Manipulating declarations in context and the goal are side effects. The support for side effects in Lean is based on the approach implemented by the Haskell programming language [30], which uses monads to isolate effectful computations from effectless ones [40]. We will not discuss deeply how this is achieved in this thesis. Instead, we will provide a high-level explanation on how to use this feature from the point of view of a Lean programmer.

In Lean, certain predetermined types, such as those built with the `IO` type constructor, are permitted to generate a particular kind of side effect associated with the type. Given a type α, a value of type `IO` α is a sequence of actions, each one possibly involving reading or writing to the filesystem or interacting with the standard input/output, where the last action must always produce a value of type α. For instance, the function `IO.FS.readFile` has type `FilePath → IO String`. Given the path of a file in the system `fp`, the application `IO.FS.readFile fp` consists of a single action, which reads all the text in the file and produces a `String`, containing the whole text. Types with such capabilities are known as *monadic types*. The restriction that the last action must always produce a value with the given underlying type is not particular to `IO`, it is a requirement for any monadic type

This class of types is also equipped with what is known as `do`-notation. This is a way of extracting values from multiple values of the same monadic type and combine them. As an example, the following code reads the contents of two files and produces the string corresponding to the concatenation of those contents:

```
def readAndConcatenate (fp1 fp2 : FilePath) : IO String := do
  let c1 : String <- IO.FS.readFile fp1
    -- c1 is now bound to the contents of the file fp1
  let c2 : String <- IO.FS.readFile fp2
    -- c2 is now bound to the contents of the file fp2
  return c1 ++ c2 -- last action producing a value of type String
```

In this work, we will be interested in two monadic type constructors: `MetaM` and `TacticM`.

Values in the family of types constructed with `MetaM` are allowed to inspect and modify the context of metavariables. Two examples of useful functions in this family are:

- **assign**: it has type `MVarId → Expr → MetaM Unit` (`Unit` is a type inhabited by a single value, normally used to indicate that the function does not produce a meaningful value) and it is used to attempt to assign the provided `Expr` to the metavariable represented by the `MVarId` received, throwing an error if the kernel indicates that the type of the `Expr` does not match the required one

- **inferType**: it has type `Expr → MetaM Expr` and, given an expression representing a well-typed term, it infers its type, returning the corresponding expression. If the input was ill-typed, it throws an error.

`TacticM` is an extension of `MetaM`. This means that any operation that can be done in `MetaM` can also be done in `TacticM`. Furthermore, `TacticM` also grants access to the current goal and to Lean's elaborator, which can compile source code into a value of type `Expr`. These two side effects enable the implementation of the function `evalTactic`, which can apply any tactic in the current context. This function is very convenient for using existing tactics in the implementation of new ones.

All tactics are functions of type `Syntax → TacticM Unit`, where `Syntax` is Lean's builtin type for representing source code. The `Syntax` received by a tactic matches exactly the code used to invoke it, and it is a responsibility of the tactic to check if this code follows the pattern that is expected.

Chapter 3

Certified Transformations

In this chapter we show how to lift proofs produced by cvc5 into proofs of Lean statements via certified transformations. First, we introduce our representation of MSFOL terms inside Lean. Then, we will show how to state and prove theorems stating relations between terms in this representation. We will define theorems matching the inference rules used by cvc5, in a way that the solver will be capable of expressing its proofs inside of Lean and have their validity checked by the kernel. Finally, we will show how to use these theorems to prove propositions originally stated in Lean.

3.1 Proof reconstruction

Consider the following snippet. It represents the encoding of the negation of modus ponens ($\neg(p \to ((p \to q) \to q))$) as an SMT problem using SMT-LIB [8], a standardized syntax for representing SMT problems:

Figure 3.1: SMT-LIB script representing the negation of modus ponens.

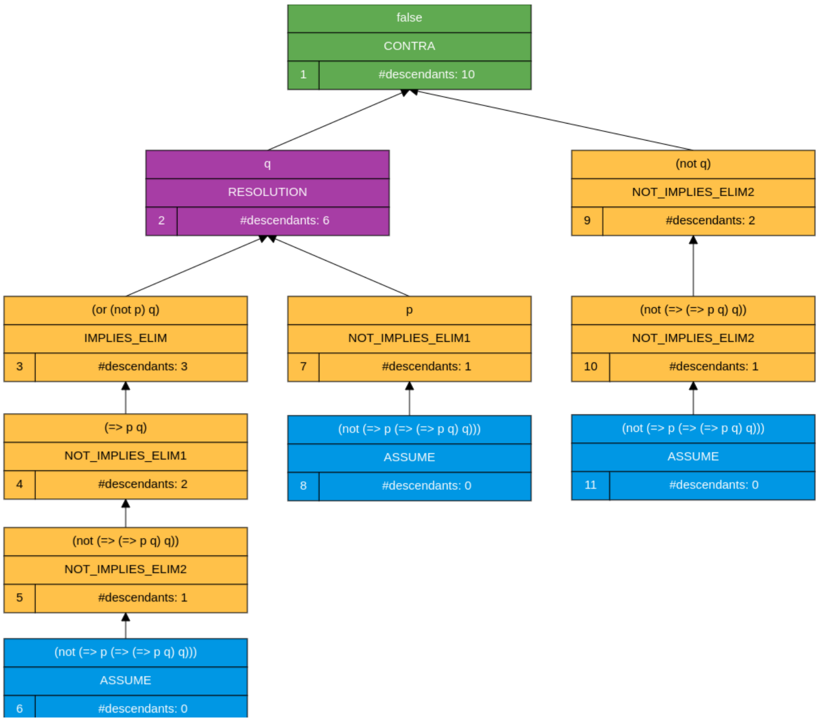
```
(set-logic QF_UF)
(declare-const p Bool)
(declare-const q Bool)
(assert (not (=> p (=> (=> p q) q))))
```

Font: Elaborated by the author.

The unsatisfability of this formula entails that its negation (modus ponens) is true for any values of p and q. Indeed, deriving the validity of a formula through the unsatisfability of its negation is a standard technique in SMT solving.

We will illustrate the process of reconstructing proofs of cvc5 via certified transformations in Lean through this example. Feeding this script to cvc5 yields the result

Figure 3.2: A cvc5 proof for the validity of Modus Ponens.



Font: https://cvc5.github.io/docs/latest/proofs/proof_rules.html

"unsat", as expected. The proof produced by the solver is shown in Figure 3.2[1] and is an instance of the resolution tree introduced in Section 2.1.3. Each node is a formula that is either the one from the input (`not (=> p (=> (=> p q) q))`) or was derived by applying some rule to other nodes. An edge from some node u to other node v indicates that u was used to derive v. In addition to the resolution rule (in purple and green), there are also rules for conjunctive normal form transformations (yellow)[2]. Note that all leaves (in blue) correspond to the input formula.

The encoding of this proof into Lean requires representing the terms that appear in it, i.e. the formulas built with p and q, as well as the rules. This representation uses a *deep embedding* of the proof calculus of cvc5 into Lean, which is done via a `term` type that models terms and formulas from MSFOL. The `const` constructor of this type can be used to define symbols, which are parameterized by an identifier (a natural number) and a `sort` (another type for enconding MSFOL sorts). For example, in the formula for modus ponens, `p` and `q` are free variables, which can be treated as constants while defining the problem in Lean. The following snippet shows their declaration using the `term` type:

```
def p: term := const 1 boolSort
def q: term := const 2 boolSort
```

[1] This was produced with the proof visualizer tool, available at: https://ufmg-smite.github.io/proof-visualizer/

[2] The rules in the figure are in the internal calculus of cvc5, which is documented in https://cvc5.github.io/docs/latest/proofs/proof_rules.html

The identifiers `1` and `2` are arbitrary, with only the requirement that they are unique.

With these terms and using the functions `implies` and `not` defined in the deep embedding corresponding to the MSFOL symbols → and ¬, we encode the formula used in the query:

```
def modusPonensEmbed: term := implies p (implies (implies p q) q)
def notModusPonensEmbed: term := not modusPonensEmbed
```

3.1.1 The Boolean fragment

We will now implement a function that lifts embedded terms to the corresponding native Lean terms, so that we can prove facts about them using Lean. Initially, we will limit ourselves to the fragment of MSFOL that deals with Boolean values[3]. Once we show how to lift this fragment, we will present a generalization of our definitions that is able to lift any theory from MSFOL.

While designing this function, one has a choice regarding which type from Lean to use as a counterpart of the Boolean terms. The two most suitable alternatives are the `Prop` and `Bool` types. The former, as previously explained, is the type used to model all propositions in the language, while the latter is the usual type of booleans inhabited by only two values: `true` and `false`. While `Bool` has a simpler structure and potentially would not require the `Classical` module (since it is possible to prove classical statements over Booleans), the extension to other theories will require, necessarily, the usage of `Prop`. For instance, consider the equality operator from MSFOL. Its applications will be mapped to applications of an equality operator in Lean. If we map MSFOL's Booleans to `Bool`, we have to map these applications to the Boolean equality, which is only defined over certain types. It is not possible to apply Boolean equality over functions, for instance. On the other hand, the equality defined over `Prop` is more general, in a way that is possible to apply it to any type.

Note that the term we will evaluate can contain free variables. For those, we will need an auxiliary interpretation function assigning concrete values to them. Free variables are identified by a `Nat` (the built-in Lean type for natural numbers), therefore, we can represent this information as a function from `Nat` to `Prop`:

```
def Interpretation := Nat -> Prop
```

[3] The Boolean fragment of the *Core* theory in SMT-LIB, as defined in `http://smtlib.cs.uiowa.edu/theories-Core.shtml`

Figure 3.3: Evaluation function.

```
def evalTerm (I : Interpretation) (t : term) : Prop :=
  match t with
  | term.const   i  _  => I i
  | term.not     t1    => Not (evalTerm I t1)
  | term.and     t1 t2 => And (evalTerm I t1) (evalTerm I t2)
  | term.or      t1 t2 => Or (evalTerm I t1) (evalTerm I t2)
  | term.implies t1 t2 => (evalTerm I t1) → (evalTerm I t2)
  | term.eq      t1 t2 => (evalTerm I t1) = (evalTerm I t2)
  | term.bot           => False
  | term.top           => True
  | _                  => False
```

Font: Elaborated by the author.

With this definition, we can define our evaluation function, which is presented in Figure 3.3. This function is matching each pattern for a `term` with the corresponding built-in operation over `Prop`, and using recursive calls of itself as arguments. If we find a `term` that is not in the fragment we are currently supporting we just return `False`. This could potentially lead to consistency issues if the input formula involved other fragments apart from Boolean. As we are limiting ourselves to this fragment for now, we will ignore this problem.

Notice how the `Interpretation` type we introduced, as well as the evaluation function, match the notions of interpretation and evaluation introduced in Section 2.1.2. Indeed, we can now define what it means for an interpretation to satisfy a `term` and what it means to be unsatisfiable:

```
def satisfies (I : Interpretation) (t : term) : Prop :=
  evalTerm I t = True
def unsatisfiable (t : term) : Prop :=
  ∀ (I : Interpretation), ¬ satisfies I t
```

One important concept we need to define is what it means for a `term` to follow logically from another, which is the primary relationship modeled by cvc5's inference rules. If, for any fixed interpretation, the evaluation of a given `term` being true implies in the evaluation of another `term` being true, then we can always conclude the second one from the first. The following definition states this relationship in Lean:

```
def impliesIn (t1 t2 : term) : Prop :=
  ∀ (I : Interpretation),
    satisfies I t1 -> satisfies I t2
```

Since we are interested in proving the unsatisfiability of terms, we will always try to prove a goal of the form `impliesIn t bot`, for some term `t`. This would imply that for any interpretation `I`, we have `(evalTerm I t = True) -> False`, provided that there is no environment that validates the interpretation of `bot`. Note that this is equivalent to `unsatisfiable t`, given our previous definition of `unsatisfiable`.

With the above we can state and prove the inference rules employed by cvc5. For instance, the following snippet shows the statement of the theorem corresponding to `notImplies1`, one of the rules of the proof in 3.2:

```
theorem notImplies1 : ∀ {t1 t2 : term},
    impliesIn (not (implies t1 t2)) t1
```

By proving the theorems corresponding to all the rules regarding Boolean terms, one can have a complete coverage of proofs produced by cvc5 (regarding the Boolean fragment of MSFOL) in Lean. Notice that, while some proofs are straightforward using case analysis (the one for `notImplies1`, for instance), some of the proofs can be quite challenging, like the one regarding the general case of Resolution. We present proofs for some of the rules in a Github repository[4].

Finally, the proof from Figure 3.2 is encoded as:

```
theorem cvc5_th0 : impliesIn notModusPonensEmbed bot :=
  fun lean_a0 =>
    have lean_s0 := notImplies2 lean_a0
    have lean_s1 := notImplies1 lean_s0
    have lean_s2 := impliesElim lean_s1
    have lean_s4 := notImplies1 lean_a0
    have lean_s6 := R1 (conjunction lean_s2 lean_s4)
    have lean_s9 := notImplies2 lean_s0
    contradiction (conjunction lean_s9 lean_s6)
```

This proof shows that the term `notModusPonens` is equivalent to `bot`, which is the same as to say that it is unsatisfiable. This way, we have encoded the proof found by cvc5 inside Lean. Also, It is possible to ask Lean's kernel to verify the proof, achieving our goal of checking proofs from cvc5 in Lean.

Note that the encoding and printing of the SMT-LIB query as a `term` is done automatically by cvc5. Therefore, we have to trust that the solver correctly printed the `term` corresponding to the conjecture it is trying to prove. However, if there was an error in the implementation of the printer that led it to incorrectly print a term, it is very

[4] https://github.com/tomaz1502/lean-smt/blob/main/Smt/Reconstruction/Certified/BooleanFragment/PropsExample.lean

likely that the proof printed would not correspond to the printed term, which would be indicated by Lean. This means that the fact that Lean accepted one of these proofs is a strong evidence that the `term` was printed correctly.

Lifting the proof to native Lean terms

Recall that our other goal was to leverage these proofs as a proof reconstruction module for a Lean hammer. Until now, everything we have shown can only be used to prove theorems stated about terms in the deep embedding, while potential users of the hammer will be interested in proving theorems phrased using Lean's native types. We will now show how to lift the proof for the unsatisfiability of modus ponens into the corresponding one stated with `Prop`. This method can easily be extended to automatically lift any proof produced by cvc5. First, we prove the following auxiliary theorem:

```
  theorem notImpliesInBot : ∀ {t : term},
    impliesIn (not t) bot → ∀ {I : Interpretation}, evalTerm I t = True
```

By applying it on the theorem generated by cvc5, we derive that, for any interpretation `I`, `evalTerm I modusPonensEmbed` is equal to `True`:

```
theorem modusPonensEqTrue: ∀ {I: Interpretation},
    evalTerm I modusPonensEmbed = True := notImpliesInBot cvc5_th0
```

Now we can define the theorem corresponding to `cvc5_th0` using `Prop` and prove it by applying `modusPonensEqTrue` to the appropriate interpretation:

```
def modusPonens (P Q : Prop) : Prop := P → (P → Q) → Q

theorem modusPonensCorrect: ∀ (P Q: Prop), (modusPonens P Q) = True := by
  intros P Q
  exact @modusPonensEqTrue (fun id => if id == 1 then P else Q)
```

where the symbol `@` is used to make explicit all parameters in the function after it. Using this interpretation, we indicate that the term `p` in `modusPonensEmbed` will be matched with the prop `P` (since the identifier of `p` is 1) and the term `q` will be matched with the prop `Q`. The checker can now compute our evaluation function and match its return value with `modusPonens P Q`, thus proving the theorem.

Therefore, we have shown how to lift the proofs produced by cvc5 into proofs that refer directly to native Lean terms. In order to fully automate this process, one has to extend cvc5's module for printing proofs. It would have to also print, for a given query, the theorem (and its proof, which is always `notImpliesInBot cvc5_th0`) corresponding to `modusPonensEqTrue` for that query, the representation of the query as a Lean term

Figure 3.4: Implementation of evalSort.

```
def evalSort : sort → Type := fun s =>
  match s with
  | arrow s1 s2 => evalSort s1 → evalSort s2
  | boolSort => Prop
  | intSort => Int
  | _ => Prop
```

Font: Elaborated by the author.

(which in this example was the term `modusPonens`) and the theorem that proves that the representation of the query is correct by instantiating the interpretation properly (corresponding to `modusPonensCorrect`).

3.1.2 Supporting other theories

Supporting more theories from MSFOL requires extending the function `evalTerm`, as well as the `Interpretation` type we defined, to be able to return values of multiple distinct types. One type safe way to achieve this in a language with dependent types is through a *sigma type*. A sigma type is a pair, in which the type of the second element depends on the value of the first element. If `T` is a type and `U` is a type constructor whose parameter is a value of type `T`, then `@Sigma T U` is the type of pairs `⟨t, u⟩` such that `t` has type `T` and `u` has type `U t`. Note that the first parameter of `Sigma` can be inferred from the second, so it is given implicitly.

Let us define a function `evalSort` that maps the type `sort` (corresponding to MSFOL's sorts in the deep embedding) to native Lean types (which are represented by `Type`). Such function is shown in Figure 3.4. We match the `arrow` sort with the arrow type used to build the type of functions, `boolSort` with `Prop` and `intSort` with `Int`. For giving support for further theories we have to extend this match statement, matching the corresponding `sort` with a suitable type. If the end goal is to be a hammer, in addition to choosing a type that shares the same properties with the sort, it is crucial to choose a type that is most commonly employed by Lean users, since the proofs produced by the ATP will have their statements expressed in terms of these types.

Now we can reformulate the `Interpretation` type, in a way that it supports any type that is also supported by `evalSort`:

```
def Interpretation := Nat → @Sigma sort evalSort
```

Figure 3.5: Reformulated interpretation used in `modusPonensCorrect`.

```
def I : Interpretation := fun id =>
  if id == 1 then ⟨ boolSort, P ⟩
  else ⟨ boolSort, Q ⟩
```

Font: Elaborated by the author.

Given an identifier, an interpretation must return a pair containing its sort and its value. Note that, with this modification, the interpretation printed by cvc5 would also need to print the sort of each term. For instance, the snippet in Figure 3.5 shows how the interpretation used in the theorem `modusPonensCorrect` would have to be rewritten.

We will also use a sigma type for defining the new version of `evalTerm`. Since we are now supporting multiple types, we have to define what is the behavior of the function if the input `term` is ill-typed. The most suitable choice is to prevent the function from returning a meaningless value, using, for instance, the `Option` type. The polymorphic `Option` type is used in Lean to indicate the possible absence of a value, which is represented by `none`, one of its constructors. The other constructor, `some`, receives, as a parameter, a single value of the type that is used as a parameter to `Option`. The alternative to this approach would be to map ill-typed terms to `False`, as we were doing with terms that were not supported in the previous version of `evalTerm`. However, this approach would introduce a logical inconsistency that would pose challenges to prove some of the rules. For instance, one rule used by cvc5 is the elimination of double negation:

```
theorem notNotElim : ∀ {t : term},
    impliesIn (not (not t)) t
```

In order to prove it, we have to consider every possible pattern for `t`. If `t` is not a valid Boolean expression, then our evaluation function would return `False` for the term `not t`, which would then force the evaluation of `not (not t)` to return `True`. Since the premise is valid in this case, we would have to prove that the conclusion is also valid, but the conclusion is not a Boolean expression.

The snippet in Figure 3.6 shows the reformulated version of `evalTerm`. We do not show the complete pattern matching for brevity, but all the other patterns are implemented using the same structure. We have also reformulated our `satisfies` predicate, in a way that it also rejects any term whose evaluation is not a `Prop`. This is shown in Figure 3.7. The predicate `impliesIn` does not require any modification. With these new definitions, one can prove the corresponding rules from any theory supported by cvc5 to enable the reconstruction of proofs from that theory in Lean.

Figure 3.6: Reformulated evaluation function.

```
def evalTerm (I : Interpretation) (t : term) :
    Option (@Sigma sort interpSort) :=
  match t with
  | term.const   i  s  =>
    let ⟨ s', value ⟩ := I i
    if s' == s then some ⟨ s', value ⟩ else none
  | term.and     t1 t2 =>
    match evalTerm I t1, evalTerm I t2 with
    | some ⟨ boolSort, p1 ⟩, some ⟨ boolSort , p2 ⟩ =>
      some ⟨ boolSort, And p1 p2 ⟩
    | _,_ => none
  | _ => none
```

Font: Elaborated by the author.

Figure 3.7: Reformulated satisfies predicate.

```
def satisfies (I : Interpretation) (t : term) : Prop :=
  match evalTerm I t with
  | some ⟨ boolSort, p ⟩ => p = True
  | _ => False
```

Font: Elaborated by the author.

3.2 Downsides of this approach

Consider the `refl` rule:

```
theorem refl : ∀ {I : Interpretation} {t : term}, satisfies I (eq t t)
```

If `t` is ill-typed, then this statement does not hold. This is not a problem for the other theorems we have shown since all of them were implications in which the term in the conclusion was a subexpression of the premise. Therefore, if the conclusion was ill-typed, the premiss was also, necessarily, ill-typed, which made the implication true. In the case of `refl` and of any other possible theorems that do not have premisses, we have to restrict it to only refer to well-typed terms in order to make their statement true.

For that purpose, we have defined a new function `inferSort` that infers the sort of a term or returns `none` if it is ill-typed. This new function is essentially identical to `evalTerm`, except that it only computes the first element of the pair. Also, since the sort of a term is independent of the interpretation we use to evaluate it, we do not need an interpretation as a parameter in this function. Now we can rephrase the statement in the theorem `refl` to make it true:

```
theorem refl' : ∀ {I : Interpretation} {t : term},
  isSome (inferSort t) = true -> satisfies I (eq t t)
```

where `isSome` is:

```
def isSome (opt : Option sort) : Bool :=
  match opt with
  | some _ => true
  | none   => false
```

The introduction of this hypothesis creates a new requirement for applying the theorem in a proof, that is, we have to provide a proof that `t` is well-typed. If `t` is well-typed (which will always be the case as long as there is no bug in the SMT solver), then Lean's kernel can evaluate both functions `inferSort` and `isSome` and obtain `true`, and the proof that it is well-typed follows by reflexivity. Deriving facts through the normalization of terms is a proof technique known as *proof by reflection* [10]. This kind of proof exchanges the cost of type checking the proofs into a cost of evaluating and matching terms.

Unfortunately, as pointed out by [2], Lean's evaluator is not optimized for normalizing terms, therefore, an excess of proofs of this kind would impact the performance of our tool. Any rule employed by cvc5 that has subexpressions in the conclusion that are not present in any of the premisses will impose the necessity of a proof by reflection of the well-typedness of these subexpressions to be formalized. There is a considerable amount of rules with this property in cvc5's calculus, therefore, this cost would result in a potential performance slowdown of the tool.

Another downside of this approach is the necessity of providing explicit proofs for all the rules. The proofs for the most complex rules can be hard to derive. For instance, the proof for the resolution rule in SMTCoq, which is also based on certified transformations, is spread out through a file with more than 700 lines.

The complexity of these proofs results in an increased rigidity in the proof certificates generated by cvc5. For instance, if the solver were optimized in the future in a manner that required modifications to the semantics of certain rules employed in the proofs, it would be necessary to prove their correctness again, which would be costly. This increase in rigidity motivated the developers of SMTCoq to implement its new features using the certifying approach [16, 15].

Furthermore, the time constraints of the project also discouraged us from pursuing this approach and deriving all these proofs.

Chapter 4

Certifying Transformations

In this chapter we describe our main contribution, which is a module for lifting an adaptation of the scripts presented in Section 3.1 into Lean proofs, based on the certifying transformations approach. The specific manner in which we applied this concept in our context was to develop a set of tactics matching the rules present in cvc5's proof calculus. A cvc5 proof is then a sequence of applications of those tactics. When we ask Lean's kernel to check the proof script, it will execute them one by one. Each one of them inspects the current context and goal and produces a proof term corresponding to a proof of a specific case of the rule represented by that tactic. The checker then verifies the correctness of each proof, closing the original goal only if all checks were succesful.

There are two options for implementing these tactics: the first one is to generate a value of type `Syntax` (that is, a piece of Lean's code) that proves the theorem in question and invoke `evalTactic` using this value; the second option is to craft an `Expr` that corresponds to a value of the appropriate type. In general, it is simpler to generate `Syntax`, as the elaborator will fill in some details for us, such as the implicit arguments for functions. On the other hand, we can skip calls to the elaborator if we produce an `Expr` instead, which will potentially lead to a gain in performance. With this in mind, we decided to employ the second option.

Another important difference from the certified transformations approach is that we do not use the `term` type anymore. The main reason for mapping MSFOL formulas to this inductive type instead of native Lean expressions is the flexibility achieved by this representation. While it is straightforward to define a function that inspects and manipulates the structure of a `term` by pattern matching, there is no way to do the same for the `Prop` type (which would be very useful to formalize the rules) without recurring to metaprogramming. As we have shown in Section 2.2.3, the metaprogramming context grants us access to the internal representation of any expression through the `Expr` type, which can be inspected and manipulated in the same way as `term`. Since the framework for writing tactics is based on metaprogramming, we do not need to rely on the flexibility of the `term` type. Therefore, we have decided to not use the deep embedding anymore, and translate MSFOL formulas directly to Lean expressions.

The snippet in Figure 4.1 shows an example of proof in the new format. It corre-

Figure 4.1: Proof script using certifying transformations.

```
1   theorem cvc5_th0 {P Q : Prop} : (Not (P → ((P → Q) → Q))) → False :=
2     fun lean_a0 : (Not (P → ((P → Q) → Q))) => by
3       have lean_s0 : (Not ((P → Q) → Q)) := notImplies2 lean_a0
4       have lean_s1 : (P → Q)              := notImplies1 lean_s0
5       have lean_s2 : (Or (Not P) Q)       := impliesElim lean_s1
6       have lean_s3 : P                    := notImplies1 lean_a0
7       have lean_s4 : Q                    :=
8         by R2 lean_s2, lean_s3, P, [1, 0]
9       have lean_s5 : (Not ((P → Q) → Q)) := notImplies2 lean_a0
10      have lean_s6 : (Not Q)              := notImplies2 lean_s5
11      exact (show False from contradiction lean_s4 lean_s6)
```

Font: Elaborated by the author.

sponds to the proof that cvc5 produces to the SMT-LIB query presented in Figure 3.1.

A considerable portion of the rules can be proved with a reasonable amount of effort. Instead of mapping such rules to tactics, we just proved them as theorems. Those theorems will not be presented here. For a complete overview of all the 75 rules, their statement and whether they were implemented with a tactic or a theorem, refer to Table A. All the rules present in the proof in Figure 4.1 were mapped to theorems except for resolution (line 8), which was mapped to a tactic that we will present.

Representation of clauses. Internally, clauses are represented by cvc5 as lists of terms, while we are representing them as disjunctions in Lean. A list of terms can correspond to many distinct disjunctions, depending on how you parenthesize them. Implicitly, cvc5 is using the fact that disjunction is associative, which implies that all of them are equivalent, therefore they do not need to be differentiated inside the calculus. When these lists of terms are sent to Lean, we are forced to chose a way to parenthesize those terms and build our proofs taking into account this format. We chose to parenthesize them in a right-associative way. Since the built-in operator ∨ is right-associative, this is the default format applied when the parenthesis are ommited. Therefore, if cvc5 is representing some clause as the list $[t_1, t_2, t_3, t_4]$ we will represent it in Lean as $\texttt{t}_1\texttt{'} \lor (\texttt{t}_2\texttt{'} \lor (\texttt{t}_3\texttt{'} \lor \texttt{t}_4\texttt{'}))$, where $\texttt{t}_i\texttt{'}$ is the representation of t_i. Throughout the implementation of the tactics, we always assume that a clause that is received as premise comes parenthesized in this way, and every clause that is generated by a tactic also has this format.

In the rest of this chapter we will give an overview of the implementation of some of the tactics. We present their statements with the same format used in cvc5's documentation[1], that is, for a tactic that has a conclusion ψ, premisses $\psi_1, \cdots, \psi_n$ and

[1] The documentation of cvc5's rules can be found at: https://cvc5.github.io/docs/cvc5-1.0.2/proofs/proof_rules.html

parameters $t_1, \cdots, t_n$, we write:

$$\frac{\psi_1, \cdots, \psi_n \mid t_1, \cdots, t_n}{\psi}$$

4.1 Auxiliary Tactics

This first class of tactics does not correspond to cvc5's rules. They were implemented to facilitate the implementation of the main tactics.

4.1.1 Parenthesizing prefixes of clauses

Given a number `i` and a proof `pf` of the validity of a clause, the application `groupClausePrefix pf, i` proves the same clause, with the first i propositions parenthesized. The disjunction of the first i propositions is a single element of the output clause. The precise statement of this rule is the following:

$$\frac{P_1 \vee \cdots \vee P_n \mid i}{(P_1 \vee \cdots \vee P_i) \vee P_{i+1} \vee \cdots \vee P_n}$$

This tactic employs the following two theorems, which were easily proven:

- `orAssoc {P Q R : Prop} : P ∨ Q ∨ R → (P ∨ Q) ∨ R`
- `congOrLeft {P Q R : Prop} (hyp : P → Q) : R ∨ P → R ∨ Q`

First, let's build a proof term for $P_1 \vee \cdots \vee (P_{i-1} \vee P_i) \vee \cdots \vee P_n$. We instantiate, in `orAssoc`, the parameter `A` to P_{i-1}, `B` to P_i and `C` to $P_{i+1} \vee \cdots \vee P_n$, obtaining the term:

$$f_1 : P_{i-1} \vee P_i \vee \cdots \vee P_n \to (P_{i-1} \vee P_i) \vee \cdots \vee P_n$$

Then, we apply `congOrLeft` to f_1 instantiating `C` to P_{i-2}, which produces:

$$f_2 : P_{i-2} \vee P_{i-1} \vee P_i \vee \cdots \vee P_n \to P_{i-2} \vee (P_{i-1} \vee P_i) \vee \cdots \vee P_n$$

We repeat this process until we get the term f_{i-1} of type $P_1 \vee \cdots \vee P_n \to P_1 \vee \cdots \vee (P_{i-1} \vee P_i) \vee \cdots \vee P_n$. Applying f_{i-1} to the original hypothesis `h` yields a term `h'` with type $P_1 \vee \cdots \vee (P_{i-1} \vee P_i) \vee \cdots \vee P_n$.

Figure 4.2: Implementation of `groupPrefixCore`.

```
1  def groupPrefixCore (pf : Expr) (i : Nat) : MetaM Expr := do
2    let clause  : Expr      ← inferType pf
3    let props   : List Expr ← collectPropsInClause clause
4    if i > 0 && i < List.length props then
5      let lemmas : List Expr ← groupPrefixLemmas props (i - 1)
6      let answer : Expr :=
7        List.foldl (fun acc lem => Expr.app lem acc) pf lemmas
8      return answer
9    else throwError
10     "[groupClausePrefix]: invalid prefix length"
```

Font: Elaborated by the author.

A clause is composed by any kind of propositions, including disjunctions. Therefore, the term $P_1 \lor \cdots \lor (P_{i-1} \lor P_i) \lor \cdots \lor P_n$ is a new clause with $n-1$ propositions, in which the $(i-1)$-th proposition is the disjunction $P_{i-1} \lor P_i$. This means that our original problem can be reduced to group the first $i-1$ terms of this clause. With this in mind, we repeat the process of composing `congOrLeft` with `orAssoc` and apply the result to `pf'`, obtaining a new term of type $P_1 \lor \cdots \lor (P_{i-2} \lor P_{i-1} \lor P_i) \lor \cdots \lor P_n$. We keep repeating this process until the whole prefix is grouped.

The snippet in Figure 4.2 shows the implementation of the core functionality of this tactic. The parameters `pf` and `i` of the function `groupPrefixCore` represent, respectively, the proof of the validity of the original clause and the length of the prefix that should be grouped. In line 2, we obtain an `Expr` corresponding to the original clause by inspecting the type of `pf`, with the built-in routine `inferType`. Then, in line 3, we use the function `collectPropsInClause` (defined by us) to extract a `List` with each one of the propositions in the clause. Next, we check if the prefix length is valid. If it is, we apply our function `groupPrefixLemmas` to obtain a list of expressions. Each element in this list is a composition of `congOrLeft` and `orAssoc` described before. One important difficulty in the implementation of this function is that implicit parameters cannot always be automatically computed in the `Expr` level, so this function also computes some of the implicit arguments that have to be passed to `congOrLeft` and `orAssoc`. Lean has a built-in functionality to automatically infer part of those arguments, but a significant amount of them have to be manually constructed. Finally, we use `foldl` in line 7 to apply each one of the lemmas to `pf`, accumulating the results.

In order to use `groupPrefixCore` as a tactic, we implement a function of type `Syntax → TacticM Unit`, which parses the `Syntax` to obtain `pf` and `i` and use the `Expr` generated by `groupPrefixCore` to close the current goal.

The following snippet shows an example of usage of this tactic:

```
theorem group3 : A ∨ B ∨ C ∨ D ∨ E → (A ∨ B ∨ C) ∨ D ∨ E := by
  intro h
  groupClausePrefix h, 3
```

We can use the `#print` command to inspect the proof term generated by the tactic, and verify that it is, indeed, the one we described:

```
#print group3 -- fun {A B C D E} h => orAssoc (congOrLeft orAssoc h)
```

4.1.2 Moving terms in clauses

Given a proof `pf` of the validity of a clause and an index `i` of a proposition to be moved, the `pull` tactic should produce a proof term for the same clause, with i-th term moved to the first position. This tactic also facilitates the implementation of the other tactics and does not exist in cvc5's proof calculus. The precise statement of this rule is the following:

$$\frac{P_1 \vee \cdots \vee P_n \mid i, s}{P_i \vee P_1 \vee \cdots \vee P_{i-1} \vee P_{i+1} \vee \cdots \vee P_n}$$

We need three theorems to implement `pull`, which are easy to prove:

- `orComm {A B C : Prop} : A ∨ B → B ∨ A`
- `congOrRight {A B C : Prop} (hyp : A → B) : A ∨ C → B ∨ C`
- `orAssocConv {A B C : Prop} : (A ∨ B) ∨ C → A ∨ B ∨ C`

To build the required proof term, let's first apply our tactic `groupClausePrefix` at `pf`, grouping the prefix of length $i - 1$. We cannot use the tactic itself, as we are working on the context of `MetaM`, but we can invoke directly the function `groupPrefixCore` to produce the required term. We will obtain the following term:

$$pf_1 : (P_1 \vee \cdots \vee P_{i-1}) \vee P_i \vee \cdots \vee P_n$$

Then, we use again our function `groupPrefixCore` in `pf`$_1$, grouping the prefix of length 2. This provides the term:

$$pf_2 : ((P_1 \vee \cdots \vee P_{i-1}) \vee P_i) \vee P_{i+1} \vee \cdots \vee P_n$$

Next, we will define a new term $\texttt{pf}_3$ as `congOrRight orComm` $\texttt{pf}_2$. Given the appropriate instantiation for the implicit parameters, this application flips the position of $P_1 \vee \cdots \vee P_{i-1}$ and P_i, while not modifying the suffix $P_{i+1} \vee \cdots \vee P_n$. Therefore, $\texttt{pf}_3$ has type $(P_i \vee P_1 \vee \cdots \vee P_{i-1}) \vee P_{i+1} \vee \cdots \vee P_n$ (we do not need to parenthesize $P_1 \vee \cdots \vee P_{i-1}$ as $\vee$ is right-associative). Finally we use another tactic we implemented, `ungroupClausePrefix`, to remove the parenthesis around $P_i \vee P_1 \vee \cdots \vee P_{i-1}$, concluding our goal. This tactic is similar to `groupClausePrefix`. The difference is that it substitutes the theorem `orAssoc` by `orAssocConv` and it applies the theorems in reverse order, i.e. it starts with an application of `orAssocConv`, then it applies `congOrLeft orAssocConv` and so on.

Disambiguating clauses. The difference between the representation of clauses in cvc5 and in Lean leads to an ambiguity in applications of this tactic. While there is no distinction in Lean between, for instance, the terms $A \vee B \vee C \vee D \vee E$ and $A \vee B \vee C \vee (D \vee E)$, this distinction can be made inside the SMT solver. Indeed, if we have a term `pf` in Lean of type $A \vee B \vee C \vee D \vee E$ and cvc5 prints the tactic application `pull pf, 4`, there is no way to know, inside Lean, if it is expecting to receive a proof of $D \vee A \vee B \vee C \vee E$ or $(D \vee E) \vee A \vee B \vee C$. To address this issue, we adapted cvc5 to print an extra parameter `s` to this tactic (and to others that suffered from the same problem), which is a natural number corresponding to the index of the last proposition in the clause from the point of view of the SMT solver. Therefore, for the example we just described, it would print `pull pf, 4, 4` to indicate that it wants a proof for $(D \vee E) \vee A \vee B \vee C$ and `pull pf, 4, 5` to indicate that it wants a proof for $E \vee A \vee B \vee C \vee D$.

4.2 Boolean reasoning

We now present the tactics regarding Boolean reasoning. One of them corresponds to the resolution rule, which as we have shown in Section 2.1.3, is the main building block used by cvc5 to produce proofs proofs in Propositional Logic. The other two, *permutateClause* and *factor*, are used to justify implicit steps (reordering of clauses and deleting repeated terms in a clause) that SAT solvers perform while solving formulas through resolution.

4.2.1 Reordering clauses

Given the proof `pf` of a clause, a permutation `perm` and the index `s` of the last proposition in the clause, the application `permutateClause pf, perm, s` provides a proof for the same clause, with the propositions permutated according to `perm`. This is the first tactic that matches a rule used by cvc5[2]. The precise statement of this rule is the following:

$$\frac{P_1 \vee \cdots \vee P_n \mid perm, s}{P_{perm(1)} \vee \cdots \vee P_{perm(n)}}$$

Since we have `pull`, we can write a tactic matching this rule using a relatively straightforward approach. We simply iterate through the permutation in reverse order, and, for each index `i` we go through, we run the `pull` tactic with `i` as the argument. With this strategy, the last term we will bring to the first position is $P_{perm(1)}$, therefore, the final clause we will produce will have the correct element in the first position. Similarly, the second to last element we will bring to the first position is $P_{perm(2)}$. After pulling it, we will bring another term to the first position ($P_{perm(1)}$), which will make $P_{perm(2)}$ end up in the second position, as required. By extending this reasoning to all the terms in the clause, we can conclude that each one of them will be placed in the right position. Notice that the index `s` of the last proposition does *not* change during this process, so we can always use `s` as the argument required by `pull`.

The code in Figure 4.3 shows our implementation of this tactic. The main function is `permutateClauseCore`. It starts by obtaining a list of `Expr` corresponding to the terms in the clause using `inferType` and `collectPropsInClause` (lines 11 and 12), in the same manner as `groupPrefixCore`. Next, in line 13, it permutates this list according to the reverse of the input permutation. Finally, in line 14, it invokes `pullProps` with the permutated list, `pf` and `s`. This auxiliary method traverses the input list, applying `pull` to each `Expr` it encounters, accumulating the result.

The following snippet shows an example of usage of this tactic:

```
theorem perm1 :
    A ∨ B ∨ C ∨ (D ∨ E ∨ F) → (D ∨ E ∨ F) ∨ B ∨ C ∨ A := by
  intro h
  permutateClause h, [3, 1, 2, 0], 3
```

[2] The documentation of this rule can be found at https://cvc5.github.io/docs/cvc5-1.0.2/proofs/proof_rules.html#_CPPv4N4cvc58internal6PfRule10REORDERINGE

Figure 4.3: Implementation of the tactic permutateClause.

```
def pullProps (props : List Expr) (acc : Expr) (s : Nat) :
    MetaM Expr :=
  match props with
    | [] => return acc
    | e::es => do
        let pulled ← pullCore e acc s
        pullProps es pulled s

def permutateClauseCore (pf : Expr) (perm : List Nat)
    (s : Nat) : MetaM Expr := do
  let clause : Expr     ← inferType pf
  let props : List Expr ← collectPropsInClause' clause s
  let permutatedProps   := permutateList props (List.reverse perm)
  pullProps permutatedProps pf s
```

Font: Elaborated by the author.

4.2.2 Resolution

Given two proofs of clauses, `pfP` and `pfQ`, a proposition `A` (also known as the *pivot*) such that `A` is an element of the first clause and `¬ A` is an element of the second clause and the indices $\mathtt{s}_1$ and $\mathtt{s}_2$ of the last propositions in each clause, this tactic produces a proof for the clause composed by the concatenation of the two, removing the pivot from the first and the negation of the pivot from the second. If there are multiple instances of the pivot in the first clause, it removes the first one. Similarly, if there are multiple instances of the negation of the pivot in the second clause, it also removes the first one. The precise statement is the following:

$$\frac{P_1 \lor \cdots \lor P_{i-1} \lor A \lor P_{i+1} \lor \cdots \lor P_n, Q_1 \lor \cdots \lor Q_{j-1} \lor \neg A \lor Q_{j+1} \lor \cdots \lor Q_m, \mid A, s_1, s_2}{P_1 \lor \cdots \lor P_{i-1} \lor P_{i+1} \lor \cdots \lor P_n \lor Q_1 \lor \cdots \lor Q_{j-1} \lor Q_{j+1} \lor \cdots \lor Q_m}$$

We will need four theorems to implement the resolution tactic, which are all easy to prove:

- `resolutionSpecialCase1 {A B C : Prop} : A ∨ B → ¬ A ∨ C → B ∨ C`
- `resolutionSpecialCase2 {A B : Prop} : A ∨ B → ¬ A → B`
- `resolutionSpecialCase3 {A C : Prop} : A → ¬ A ∨ C → C`
- `resolutionSpecialCase4 {A : Prop} : A → ¬ A → False`

First, we apply `pull` to `pfP` to bring the first occurence of `A` to the first position. This will result in a proof `pfP’` of the clause:

$$A \lor P_1 \lor \cdots \lor P_{i-1} \lor P_{i+1} \lor \cdots \lor P_n$$

Next, we do the same at `pfQ`, bringing the negation of the pivot to the first position. We will obtain a proof `pfQ’` of the clause:

$$\neg A \lor Q_1 \lor \cdots \lor Q_{j-1} \lor Q_{j+1} \lor \cdots \lor Q_m$$

Now, we apply one of the `resolutionSpecialCase` theorems to `pfP’` and `pfQ’`. We decide which theorem to apply based on whether or not each clause has other terms apart from the pivot. If both clauses have other terms (i.e. $n > 0$ and $m > 0$) we apply `resolutionSpecialCase1`, instantiating `B` to $P_1 \lor \cdots \lor P_{i-1} \lor P_{i+1} \lor \cdots \lor P_n$ and `C` to $Q_1 \lor \cdots \lor Q_{j-1} \lor Q_{j+1} \lor \cdots \lor Q_m$. If the second clause consists of only the negation of the pivot and the first one has other terms, we apply `resolutionSpecialCase2`. If the first clause consists of only the pivot and the second one has other terms, we apply `resolutionSpecialCase3`. If both clauses consist of a single element, we apply `resolutionSpecialCase4`. In this case, the tactic will produce a proof of `False`, which is the way we represent the empty clause.

Notice that, if we apply `resolutionSpecialCase1` and the first clause had more than 1 element apart from the pivot, the term we get is not the required one. Its type will have the propositions from the first clause parenthesized, i.e. it will have the following form:

$$(P_1 \lor \cdots \lor P_n) \lor Q_1 \lor \cdots \lor Q_m$$

We fix this issue with the tactic `ungroupClausePrefix`, which was mentioned before.

The original rule used by cvc5 has another boolean parameter, *pol*. If pol is set to true, then the semantics of the rule matches exactly the tactic we described. If pol is set to false, the rule expects to find the pivot negated in the first clause and not negated in the second. We have implemented a separate tactic for this case. The function that implements the core functionality of both tactics is the same.

The following snippet shows one example of usage of each tactic. `R1` corresponds to the rule with pol set to true, and `R2` corresponds to the rule with pol set to false. The last parameter of the tactic is a list with two elements, corresponding to the indices of the last propositions in each clause.

```
theorem res1 : A ∨ B ∨ C ∨ D → E ∨ ¬ B → A ∨ (C ∨ D) ∨ E := by
  intros h₁ h₂
  R1 h₁, h₂, B, [2, 1]

theorem res2 : ¬ A → B ∨ A ∨ C → B ∨ C := by
  intros h₁ h₂
  R2 h₁, h₂, A, [0, 2]
```

4.2.3 Removing duplicates

Given a proof `pf` of a clause and the index `s` of the last proposition in the clause, the *factor* tactic produces a proof of the same clause, with all duplicated literals of it removed. The precise statement of this rule is the following:

$$\frac{P_1 \vee \cdots \vee P_n \mid s}{removeDuplicates(P_1 \vee \cdots \vee P_n)}$$

We will need two theorems to implement it:

- `dupOr {A B : Prop} : A ∨ A ∨ B → A ∨ B`
- `dupOr' {A : Prop} : A ∨ A → A`

Also, in order to implement factor, we employ the tactic `pullToMiddle`, a generalization of `pull`. It allows its user to move a term in position j in some clause to any position $i < j$. The implementation of `pullToMiddle` is similar to the one for `pull`, except that instead of grouping and ungrouping prefixes of the clause, it considers intervals in the middle of the clause.

The idea we employed to develop this tactic is the following: for each literal in the clause, we check every other literal to see if it is equal to the current one. If it is, we obtain a new clause with the literals that are equal adjacent to each other. We then apply either `dupOr` or `dupOr’` (depending whether they are on the last positions of the clause), together with the appropriate congruence lemmas (like we did for grouping clauses). We repeat this process until there are no more duplicates in the clause.

The pseudocode in Figure ?? shows our implementation of this idea. First, in line 1, we obtain the clause corresponding to `pf` using `inferType`, in the same way we did in the implementation of `groupClausePrefix`. Then, in lines 2 to 15 iterate through each term in the clause. We use the function *GetLength* to obtain the current length of the clause. We need to provide the index of the last proposition to this function, otherwise the length of the clause will not be well defined. The goal of each iteration

```
1:  function FactorCore($pf$, $s$)
2:      $clause \leftarrow$ InferType($pf$)
3:      for $i \leftarrow 1$ up to GetLength($clause$, $s$) do
4:          $j \leftarrow i + 1$
5:          while $j <$ GetLength($clause$, $s$) do
6:              if $clause_i = clause_j$ then
7:                  $pf \leftarrow$ PullToMiddleCore($pf$, $i + 1$, $j$, $s$)
8:                  $pf \leftarrow$ ApplyDupOr($pf$, $i$, $i + 1$)
9:                  $s \leftarrow s - 1$
10:                 $clause \leftarrow$ InferType($pf$)
11:             else
12:                 $j \leftarrow j + 1$
13:             end if
14:         end while
15:     end for
16:     return $pf$
17: end function
```

is to remove all other elements of the clause that are equal to the i-th one (the current one). For that, we do another nested loop in lines 5 to 14, iterating through all indices j such that $j > i$. Next, in line 6, we check whether the propositions at positions i and j are equal (we use the notation $clause_i$ to indicate the proposition at position i). This comparison is syntactic, that is, we only consider equal propositions that have *exactly* the same representation as `Expr`. This is the intended behaviour of this rule. It is designed to remove duplications introduced by applications of resolution (which will be syntactically equal). If the propositions are different, we simply increment j. Otherwise, we bring the j-th proposition to position $i+1$ using the tactic `pullToMiddle`. This will allow us to use one of the `dupOr` theorems. If $i + 1$ is the last position in the clause, we apply `dupOr'`, otherwise we apply `dupOr`. Notice that, since there are potentially other terms to the left of the i-th one, we need to apply `congOrLeft` composed with `dupOr`, in the same way we did for `groupClausePrefix`. The function *ApplyDupOr* performs these checks and applies the correct version of `dupOr`, producing a new proof term for the clause with the duplicate erased. Since the number of elements to the left of the last one necessarily decreased by one, we have to decrement s, which is done in line 9. Finally, we update *clause* according to our modifications by extracting again the type of pf.

Notice that the outer loop maintains following invariant: during the i-th iteration, all propositions on the i-th prefix are distinct. Therefore, it is not necessary to check propositions with index lesser than i.

The following snippet shows an application of this tactic:

```
theorem factor1 :
    A ∨ B ∨ (E ∨ F) ∨ B ∨ A ∨ (E ∨ F) → A ∨ B ∨ (E ∨ F) :=
by intro h
   factor h, 5
```

Here, the tactic was used to eliminate the second occurrences of `A`, `B` and `(E ∨ F)` at the hypothesis `h`.

4.3 Linear Arithmetic reasoning

In this section we present three tactics regarding Linear Arithmetic reasoning. The first two, `sumBounds` and `arithMulPos`/`arithMulNeg`, are used to derive false by combining the inequalities in the problem scaled by the Farkas' coefficients, as we explained in Section 2.1.3. The third one, `tightBounds`, is the justification for a transformation (tightening bounds based on the fact that a variable is an integer) performed by solvers of Linear Integer Arithmetic.

4.3.1 Summing lists of inequalities

Given a list of n proofs of statements following one of the patterns $a_i < b_i$, $a_i \leq b_i$ or $a_i \simeq b_i$, the tactic `sumBounds`[3] produces a proof of the inequality $\sum_{i=1}^{n} a_i \bowtie^* \sum_{i=1}^{n} b_i$, where $\bowtie^*$ is $<$ if all terms in the premisses are strict inequalities and $\leq$ otherwise. Its precise statement is the following:

$$\frac{\bigwedge_{i=1}^{n} a_i \bowtie_i b_i}{\sum_{i=1}^{n} a_i \bowtie^* \sum_{i=1}^{n} b_i}$$

Representing Int and Real. Each one of the variables a_i and b_i might have sort either *Int* or *Real*. While each pair a_i and b_i always shares the same sort, it is possible that some of these pairs have a sort different from the rest. Before implementing a tactic corresponding to this rule, we have to choose types in Lean to match these two sorts. We have based our decision on the types used by mathlib, as individuals formalizing new mathematical statements in this library are one of the main potential users of a Lean hammer. Integers are represented in mathlib exclusively with the built-in type *Int*, so

[3] The documentation for the corresponding cvc5 rule can be found at https://cvc5.github.io/docs/cvc5-1.0.2/proofs/proof_rules.html#_CPPv4N4cvc58internal6PfRule12ARITH_SUM_UBE

we will use it to represent the Int sort. For the Real sort, we could either represent it using the type `Real`, which was defined in mathlib to represent real numbers, or the type `Rat`, which was defined in the package *std4*[4] to represent rational numbers and is used by mathlib to formalize a variety of theorems that involve this type of number. It is not a problem to use rational numbers to represent the Real sort here since the rules employed by cvc5 in the theory of linear arithmetic do not explore any particular property that is enjoyed by real numbers and not by rational numbers. We decided to employ the type `Rat`, as its definition is much simpler and easier to manage in comparison to the one for `Real`.

The implementation of this tactic requires 9 variations of the following theorem:

```
sumBoundsThm {α : Type} [LinearOrderedRing α] {a b c d : α} :
  a < b → c < d → a + c < b + d
```

Each variation corresponds to one combination of the relation symbols in the hypothesis, where they can be either $<$, $\leq$ or $\simeq$. The relation symbol in the conclusion is adapted accordingly in each theorem. Since the rule accepts mixing of variables from Int and Real sort, we need a variation of each one of those 9 theorems for each combination of the types of the variables. Instead of stating all the combinations explicitly, which would result in a total of 36 theorems and a long branch in the implementation of the tactic, we stated only one polymorphic version of each, as indicated by the type parameter α in `sumBoundsThm`. Obviously, the theorem does not hold for any α (it cannot even be stated if there is no comparison and addition operators defined over α). We solve this issue by adding a restriction, stating that α satisfies the axioms of a *Linear Ordered Ring* (a class of types that contains both `Int` and `Rat` defined in mathlib), represented by the expression `[LinearOrderedRing α]` in the theorem. With this restriction we could find a proof for each theorem.

The idea we employed to develop this tactic is the following: we set the last proof in the input as an accumulator. Then, we iterate through the rest of the proofs in reverse order. For each proof we go through, we produce a new term, adding the inequality represented by this proof with the one in the accumulator. This is done through one of the `sumBoundsThm` theorems. Lean's simplifier always moves the parenthesis of the addition notation to the right, therefore, the proof term we are producing must prove an inequality in which the additions are also associating to the right. We can obtain such a proof term by iterating in reverse order and always adding the new terms to the left of the inequality represented by the accumulator.

Figure 4.4 shows our implementation of this idea. The function `sumBoundsCore` is supposed to be invoked with the last proof in the input as `acc` and the rest of them, in reverse order, as `pfs`. If `pfs` is empty, we just return our accumulator (line 18). Otherwise,

[4] std4 is Lean's standard library. It can be accessed at: https://github.com/leanprover/std4

Figure 4.4: Implementation of the `sumBounds` tactic

```
1  def combineBounds (pf₁ pf₂ : Expr) : MetaM Expr := do
2    let t₁ ← inferType pf₁
3    let t₂ ← inferType pf₂
4    let rel₁ ← getRel t₁
5    let rel₂ ← getRel t₂
6    let opType₁ ← getOperandType t₁
7    let opType₂ ← getOperandType t₂
8    let (pf₁', pf₂') ← castHypothesis pf₁ pf₂ rel₁ rel₂ opType₁ opType₂
9    let thmName : Name :=
10     match rel₁, rel₂ with
11     | `LT.lt , `LT.lt => `sumBoundsThm₁
12     | `LT.lt , `LE.le => `sumBoundsThm₂
13     | _      , _      => panic! "[sumBounds]: invalid relation"
14   mkAppM thmName #[pf₂', pf₁']
15
16 def sumBoundsCore (acc : Expr) (pfs : List Expr) : MetaM Expr :=
17   match pfs with
18   | [] => return acc
19   | pf' :: pfs' => do
20     let acc' ← combineBounds acc pf'
21     sumBoundsCore acc' pfs'
```

Font: Elaborated by the author.

we invoke our function `combineBounds` to sum the inequalities represented by `acc` and `pf'` (line 20) and recursively call `sumBoundsCore`, updating the arguments (line 21). The first action performed in function `combineBounds` is to obtain the inequalities corresponding to `pf`$_1$ and `pf`$_2$ through `inferType`. Then, in lines 4 to 7, it inspects their structure and obtains their relation symbol (`rel`$_1$ and `rel`$_2$) and the type of their operands (`opType`$_1$ and `opType`$_2$). Notice that `sumBoundsThm` expects that all the four variables have the same type. If one of the input proofs represents an inequality over integers and the other is over rationals, we have to lift the inequality between integers into an inequality between rationals. This is done using one of the following theorems, depending on the relation symbol:

- `Int.castLT : ∀ {a b : Int}, a < b → Rat.ofInt a < Rat.ofInt b`
- `Int.castLE : ∀ {a b : Int}, a ≤ b → Rat.ofInt a ≤ Rat.ofInt b`
- `Int.castEQ : ∀ {a b : Int}, a = b → Rat.ofInt a = Rat.ofInt b`

where `Rat.ofInt` is the standard function to cast an integer into a rational. The function `castHypothesis` (line 8) does this analysis and, if necessary, applies one of these theorems to $\texttt{pf}_1$ or $\texttt{pf}_2$. Once we have the correct version of the proofs, we can apply one of the versions of `sumBoundsThm`, depending on $\texttt{rel}_1$ and $\texttt{rel}_2$. The `match` statement in lines 10 to 13 chooses which version to apply. We only show two cases for brevity. The backtick is used to transform a literal into a value of the built-in type `Name`, that is employed by Lean to represent identifiers. Finally, in line 14, we apply the chosen theorem to $\texttt{pf}_2\texttt{'}$ and $\texttt{pf}_1\texttt{'}$. We have to invert the order here to compensate the fact that we are using these proofs in reverse order, compared to the order they came in the input. The function `mkAppM` is a built-in function from the metaprogramming framework, which tries to infer the implicit arguments for a given function application. In this case, it succeeds, since they can be inferred from the type of $\texttt{pf}_1\texttt{'}$ and $\texttt{pf}_2\texttt{'}$.

4.3.2 Scaling inequalities

Given the variables `m`, `l` and `r` and an index `id`, we present two tactics for multiplying an inequality between `l` and `r` by `m`. The parameter `id` is used to indicate what is the inequality between `l` and `r` in question (lesser than, lesser or equal, greater than, or greater or equal). The `m` argument passed to this tactic is precisely the Farkas' coefficient calculated by cvc5. It is necessary since the rule[5] corresponding to this tactic does not receive the proof of the inequality as a premise, instead, it produces an implication from such an inequality to its scaled version. The first tactic, `arithMulPos`, assumes that `m` is positive. Its precise statement is the following:

$$\frac{- \mid m, l, r, id}{(m > 0 \land l \bowtie_{id} r) \to m \cdot l \bowtie_{id} m \cdot r}$$

The second tactic, `arithMulNeg`, assumes that `m` is negative. Its precise statement is the following:

$$\frac{- \mid m, l, r, id}{(m < 0 \land l \bowtie_{id} r) \to m \cdot l \bowtie_{inv(id)} m \cdot r}$$

where $\bowtie_{inv(id)}$ is $>$ if $\bowtie_{id}$ is $<$, $\geq$ if $\bowtie_{id}$ is $\leq$ and so on.

We could have just proved a theorem for this rule if every variable had the same type, but this is not the case. Each one of them (`m`, `l` and `r`) can be either an `Int` or a `Rat`. This tactic inspects the type of each one of the variables and apply one of the eight corresponding theorems, accordingly, in a manner similar to `sumBounds`. All these theorems are easy to be proved.

[5] Its documentation can be found at https://cvc5.github.io/docs/cvc5-1.0.2/proofs/proof_rules.html#_CPPv4N4cvc58internal6PfRule14ARITH_MULT_POSE

4.3.3 Tightening bounds

Given an integer `i`, a numeric variable `q` and a proof of a strict inequality between `i` and `q`, the pair of tactics `tightLb` and `tightUb` produce proofs of a tighter relation between `i` and `q`. More specifically, the following are the precise statements of `tightLb` and `tightUb`, respectively:

$$\frac{i < q \mid i, q}{i \leq \lceil q \rceil - 1}$$

$$\frac{i > q \mid i, q}{i \geq \lfloor q \rfloor + 1}$$

Once again, this tactic could be just a theorem, except that cvc5 can apply it with variables from both Real and Int sorts as the parameter `q`. The logic of this tactic consists of checking what is the type of `q` and applying the appropriate theorem.

Unlike the previous theorems, this one was not easy to prove. In fact, the implementations of the *ceil* and *floor* functions (from the package *std4*) we were using while developing this tactic were incorrect[6]. When dealing with negative values, they were outputting a value one less than the correct output. We defined and used our own versions of these functions to prove the theorems, and, when the version of the library was fixed, we substituted our version by it. Our original proofs had almost 350 lines combined, which was a consequence of the fact that there were almost no theorems in mathlib about these functions. Once more theorems were added to the library, we could reduce each proof to 15 lines.

4.4 Equality reasoning

As we have shown in Section 2.1.3, proof certificates for the theory of Equality and Uninterpreted functions consist of proof trees, where the inference rules are the axioms of equality (reflexivity, symmetry, transitivity and congruence). Lean already has built-in theorems corresponding to versions of these axioms that refer to the built-in equality operator, which is the one that we are mapping MSFOL equalities to. Hence, there was no need to introduce any new theorems or tactics for this theory.

[6] The pull request that fixed them is the following: https://github.com/leanprover/std4/pull/80

4.5 Comparison with Certified Transformations

The main advantage of the certifying approach is the fact that it is, in general, easier to write a tactic that generates a proof on the fly for an application of a rule than to prove the correctness of the rule considering all cases. While proving these theorems for some of these rules (such as permutateClause and factor) appears to be a hard task, the implementations of the tactics are not too complicated (the complete implementation of `permutateClause`, for instance, consists of fewer than 60 lines). This also means that we can modify the semantics of a rule without too much effort. This flexibility was explored for instance at [43], where the authors incremented an already existing framework of proof reconstruction based on certifying transformations to support a novel format of proof certificates produced by a newly introduced SMT solver.

Moreover, unlike the certified transformations approach, the one we are introducing now does not rely on the normalization of terms to prove statements. There is, however, a novel overhead of checking the proofs produced by the tactics each time the hammer is executed. Due to the results obtained at [2], we believe that this overhead is less significant than the one introduced by the normalization of terms, but it would be necessary to conduct experiments to confirm this hypothesis. We could not run such experiments since we did not had the time to fully implement the certified approach.

Another indicative of the potential of the certifying approach is the success of Sledgehammer, the corresponding tool for Isabelle/HOL, which is also based on this technique. In [14], the authors present a benchmark executed with 1240 theorems proved in Isabelle's library. They were drawn from the formalizations of various complex concepts in the ITP, including the Fast Fourier Transform [19] and the Fundamental Theorem of Algebra. They executed the hammer on each one of them, without human interaction. For 60.1% of them, the hammer was able to find and reconstruct a proof within 90 seconds, which is a promising outcome, considering that the theorems were not trivial.

Chapter 5

Evaluation

In this chapter we will present the evaluation of our set of tactics. The main focus of our experiments is to measure the coverage of our proof reconstruction library. More concretely, we want to check how often our framework, when integrated with cvc5's proof-generation mechanisms, is capable of reconstructing the solver's reasoning. We will investigate whether there are any mismatches between the semantics of some inference rule and the corresponding tactic, as well as assess cvc5's capability of completely expressing its reasoning through proof certificates.

As we have explained before, we do not have as a goal to implement a hammer for Lean that would be as efficient as SMTCoq or Sledgehammer. Considering the time constraints of our project and the fact that Lean currently does not have any project that attacks this problem, our goal is limited to determine whether it is possible to perform proof reconstruction using cvc5's output in Lean, and how to do it. We believe that this is an important first step towards having a hammer for Lean that is fast and powerful. With this in mind, we will not compare the performance of our tool with these established hammers.

5.1 Strategy

In order to evaluate our tool, we used the benchmark provided by the SMT-LIB initiative [7]. The problems in this benchmark are categorized by the combination of theories they are expressed in. Given our support for the theories of Equality and Uninterpreted Functions, Linear Integer Arithmetic and Linear Real Arithmetic, we can reconstruct problems from the combinations of theories identified in the benchmark by QF_LIA, QF_LIRA, QF_LRA, QF_UF, QF_UFLIA, QF_UFLRA (QF stands for quantifier free and is used to identify problems that do not have quantifiers in their definition). Ideally, we would be able to reconstruct any proof for a query in this set of theories. SMTCoq and Sledgehammer would be able to reconstruct proofs in the same theories and in the theory of Bitvectors. For each problem within this set, we executed a version of cvc5 that has the experimental feature of printing its proofs as Lean scripts[1]. We gave it a timeout of 600s for each problem to determine which ones were unsatisfiable and generate a proof[2] for their unsatisfiability, resulting in a total of 6102 proofs. From our experience, proofs exceeding 1MB are too costly to be checked with the current state of our framework, therefore, we have filtered out the proofs that surpassed this threshold, which amounted to a total of 876 proofs.

For the purpose of checking these proofs, we have written a Lean script that, given a Lean file, loads our framework and checks if all the proofs contained in the file are correct[3]. It does not check whether the theorem in Lean indeed corresponds to the original SMT query. As we have explained before, if the translation from the query to the theorem was wrong, it is very likely that the translated proof would also be wrong, so the correctness of the proof is a strong evidence for the correctness of the translation.

Using the binary produced when our script is compiled, one can reconstruct the proofs without installing Lean. It just requires that all the object files from our library (and from its dependencies) are in the same folder as the binary. Notice that this script has to load all the libraries in each run, which heavily impacts its performance. We emphasize again that our goal with this evaluation is to test the coverage of our library and not its performance. Writing a tool for proof reconstruction that prioritizes performance requires giving more attention to different aspects of the system and is another project in itself. All experiments were run on a server equipped with 32 processors Intel(R) Xeon(R) CPU E5-2620 v4 2.10GHz and 125.79 GiB RAM, running Ubuntu 20.04, kernel 5.4.0-132-generic, with 8GB of memory for each job.

5.2 Analysis

The results we obtained are presented in Table 5.1, categorized by SMT-LIB logic. From a total of 876 proofs, our tool could reconstruct successfully 281 of them. The reconstruction of the other 596 proofs failed for the following reasons (note that some errors occurred in more than one test case):

1. 89 failed due to steps asserting equalities between terms of different types, which is

[1] We used the version of cvc5 from this commit: https://github.com/HanielB/cvc5/tree/b2340f42639733a0ef9523aee2b68f0bf062a5a7

[2] It is necessary to pass the following flags to cvc5 to generate the proofs as Lean scripts: `--dump-proofs --dag-thresh=0 --proof-granularity=theory-rewrite --proof-format=lean`

[3] The script can be found at https://github.com/tomaz1502/process_lean_smt/blob/main/Main.lean

Table 5.1: Result of proof checking per theory.

Theory	Total	Succeeded	Error while checking	Timeout	Memout
QF_UF	299	64	117	9	109
QF_LIA	236	30	161	0	45
QF_LRA	205	91	110	4	0
QF_UFLIA	106	67	30	9	0
QF_UFLRA	31	29	2	0	0

Font: Elaborated by the author.

allowed in cvc5's calculus (since it supports subtyping) but is not allowed in Lean's type system.

2. 109 failed due to cvc5 assuming that Lean's `Ne` operator is n-ary, while it is binary.

3. 5 failed because cvc5 was implicitly reordering the operands of an addition or a disjunction.

4. 22 failed for exceeding the limit of time.

5. 154 failed for exceeding the limit of memory.

6. 125 failed for exceeding the limit of resources for a single tactic execution (we have set `maxNumHeartBeats` to 500000).

7. 11 failed since the tactics `arithMulPos` and `arithMulNeg` did not suppport mixing of integers and rations between its parameters `l` and `r`.

8. 28 failed because Lean incorrectly inferred the type of some term in an intermediate congruence step.

9. 75 failed since Lean can't automatically lift an equality between integers to an equality between rationals, which is implicitly done by cvc5.

After running the benchmark, we fixed issue 7 by adapting the tactic to perform the appropriate castings if the terms passed as arguments had different types. We also fixed issues 8 and 9 by replacing the congruence theorem with a tactic that uses Lean's rewriter to simulate the congruence rule. This change has two advantages: first, since the rewrite is capable of coercing the types when needed, we can skip the process of matching the types of the hypothesis with the types expected by the congruence theorem, which will eliminate the need for type inferences that were previously done incorrectly; second, Lean's rewriter can automatically lift an equality between integers into an equality between rationals if needed, solving the last problem. This is needed, for instance, if we have an hypothesis `(0 : Int) = b` and cvc5 wants to use it to prove `a + b = a + 0`, where `a` is a rational. All these fixes were tested and are working properly.

Issue 1 should be fixed inside cvc5's proof printing module, by adding the appropriate type coercions each time an equality relying on subtyping would be built. Issue 2 can be fixed either in Lean, through the introduction of an n-ary inequality operator, or in cvc5, by transforming the n-ary inequality into a conjunction of multiple inequalities. Issue 3 should also be fixed inside cvc5. It is necessary to add intermediary steps to reorder the additions and disjunctions before using them. Disjunctions can be reordered using our `permutateClause` tactic and additions can be reordered using mathlib's `linarith` tactic.

The results of the benchmark point out that, despite its performance issues, our tool can cover most of the reasoning performed by cvc5 for these logics. Within the issues indicated by our set of test cases, only the first three remain as covering problems, and they should be solved by adapting cvc5's printer. This outcome makes clear that the main focus of the future work should be on optimizing the performance of the tool. In Chapter 6, we will show some promising ideas we have for improving it.

Chapter 6

Future Work

6.1 Skipping Lean's parser

The main bottleneck on the performance of our tool is the parsing of the proof by Lean. The flag `--profile` from Lean's compiler can be used to indicate how much time was spent on each process during the checking of a proof. The tests we did with this flag pointed out that over 90% of the time was spent during parsing. Therefore, if the Lean's parser were skipped we could increase the speed of the reconstruction of proofs.

This could be done by modifying cvc5's printer, so that it prints terms in an intermediate format that is similar to the internal representation used by Lean (i.e. with the type `Expr`). Then, the proof format would be changed to a list containing the steps of the proof. Each step would be represented by the name of a theorem or tactic, the arguments passed to it and the expected return type, printed in this new intermediate format. Also, it would be necessary to implement a lightweight parser in Lean for this proof format. Clearly, this parser would be much simpler than Lean's. Since our tactics are implemented using the `MetaM` monad, this optimization would completely remove any calls to Lean's parser in our tool.

6.2 Naming shared terms

Another factor that contributes to reduce performance is the size of the proof scripts. Frequently, the same expression is printed multiple times across the proofs. One simple way to reduce the size of the scripts is to define macros for any expression whose abstract syntax tree has a depth greater than 1 and that appear multiple times. By printing the macros instead of the expressions themselves, the proof will not have any term whose depth is greater than 1, potentially leading to an exponential reduction in its size.

6.3 Parsing other proof formats

Currently, our framework for proof reconstruction only supports proofs generated in the format produced by cvc5 that we have shown. Therefore, it can only verify cvc5 proofs. By implementing a parser for other proof formats (such as the Alethe format [42]) we could reconstruct proofs generated by any solver that supports these formats. Note that it would also be necessary to either add new tactics in our library matching the inference rules used in the target proof calculi, or finding a way to reproduce these rules with our tactics. This would also increase the potential of the tool as a hammer, since it will enable the possibility of using different solvers, which have different strengths, to search for proofs for a given goal.

6.4 Supporting new theories

We have implemented proof reconstruction for Boolean, Equality and Linear Arithmetic reasoning. SMT solvers also support a variety of other theories that would be interesting to include in our framework. One interesting addition would be the theory of *Bitvectors*, that is, arrays of bits, which capture the semantics of machine integers and are ubiquitous on formalizations of implementations of algorithms and data structures. Also, some SMT solvers (in particular, cvc5 [31]) support reasoning over formulas involving quantifiers. This kind of formulas are much more expressive than the ones we are currently supporting. Moreover, theorems proven in ITPs frequently involve quantifiers. Therefore, adding support for quantifier reasoning in our framework would be very useful.

Supporting these theories would require writing new theorems and tactics matching their rules and extending the printer to use these theorems and tactics in the produced scripts.

6.5 Performance evaluation

Once the tool is improved, we also intend to do a performance evaluation of it, following the format done by [1]. In particular, we would like to analyze which tactics are being used more frequently and should be targets for optimization. It would also be interesting to compare our tool to one that just performs proof checking independently of an ITP, such as [1]. This would provide an accurate parameter for evaluating whether our tool has a good performance.

Bibliography

[1] Bruno Andreotti, Hanna Lachnitt, and Haniel Barbosa. Carcara: An efficient proof checker and elaborator for SMT proofs in the alethe format. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part I*, volume 13993 of *Lecture Notes in Computer Science*, pages 367–386. Springer, 2023.

[2] Anne Baanen. A lean tactic for normalising ring expressions with exponents (short paper). In *Automated Reasoning: 10th International Joint Conference, IJCAR 2020, Paris, France, July 1–4, 2020, Proceedings, Part II*, page 21–27, Berlin, Heidelberg, 2020. Springer-Verlag.

[3] 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.

[4] 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.

[5] Henk Barendregt and Erik Barendsen. Introduction to Lambda Calculus. March 2000.

[6] 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.

[7] Clark Barrett, Leonardo de Moura, Silvio Ranise, Aaron Stump, and Cesare Tinelli. The smt-lib initiative and the rise of smt. In Sharon Barner, Ian Harris, Daniel Kroening, and Orna Raz, editors, *Hardware and Software: Verification and Testing*, pages 3–3, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[8] 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.

[9] Paul Bernays. W. ackermann. solvable cases of the decision problem. studies in logic and the foundations of mathematics. north-holland publishing company, amsterdam1954, viii 114 pp. *The Journal of Symbolic Logic*, 22(1):68–72, 1957.

[10] Yves Bertot and Pierre Castéran. ** Proof by Reflection*, pages 433–448. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[11] 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.

[12] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

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

[14] 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.

[15] Valentin Blot, Denis Cousineau, Enzo Crance, Louise Dubois de Prisque, Chantal Keller, Assia Mahboubi, and Pierre Vial. Compositional pre-processing for automated reasoning in dependent type theory. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP '23. ACM, January 2023.

[16] 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.

[17] 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.

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

[19] James Cooley and John Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

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

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

[22] 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.

[23] 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.

[24] Bruno Dutertre and Leonardo Mendonça de Moura. Integrating simplex with dpll(t). 2006.

[25] 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.

[26] Bernard A. Galler and Michael J. Fisher. An improved equivalence algorithm. *Commun. ACM*, 7(5):301–303, may 1964.

[27] Georges Gonthier. The four colour theorem: Engineering of a formal proof. In Deepak Kapur, editor, *Computer Mathematics, 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers*, volume 5081 of *Lecture Notes in Computer Science*, page 333. Springer, 2007.

[28] Thomas Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. A formal proof of the kepler conjecture, 2015.

[29] William Alvin Howard. The formulae-as-types notion of construction. In Haskell Curry, Hindley B., Seldin J. Roger, and P. Jonathan, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980.

[30] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language haskell: A non-strict, purely functional language version 1.2. *SIGPLAN Not.*, 27(5):1–164, may 1992.

[31] Mikolás Janota, Haniel Barbosa, Pascal Fontaine, and Andrew Reynolds. Fair and adventurous enumeration of quantifier instantiations. *CoRR*, abs/2105.13700, 2021.

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

[33] 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.

[34] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 625–635, Cham, 2021. Springer International Publishing.

[35] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, apr 1980.

[36] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM*, 53(6):937–977, nov 2006.

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

[38] Christine Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. In Bruno Woltzenlogel Paleo and David Delahaye, editors, *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic (Mathematical logic and foundations)*. College Publications, January 2015.

[39] Arthur Paulino, Damiano Testa, Edward Ayers, Evgenia Karunus, Henrik Böving, Jannis Limperg, Siddhartha Gadgil, and Siddharth Bhat. A lean 4 metaprogramming book. GitHub. URL: https://github.com/leanprover-community/lean4-metaprogramming-book (visited on 2023-08-23).

[40] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, page 71–84, New York, NY, USA, 1993. Association for Computing Machinery.

[41] A. Schrijver. *Theory of Linear and Integer programming.* Wiley-Interscience, 1986.

[42] Hans-Jörg Schurr, Mathias Fleury, Haniel Barbosa, and Pascal Fontaine. Alethe: Towards a generic SMT proof format (extended abstract). In Chantal Keller and Mathias Fleury, editors, *Proceedings Seventh Workshop on Proof eXchange for Theorem Proving, PxTP 2021, Pittsburg, PA, USA, July 11, 2021*, volume 336 of *EPTCS*, pages 49–54, 2021.

[43] Hans-Jörg Schurr, Mathias Fleury, and Martin Desharnais. Reliable reconstruction of fine-grained proofs in a proof assistant. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 450–467, Cham, 2021. Springer International Publishing.

[44] A. Stump, A. Reynolds, C. Tinelli, A. Laugesen, Harley Eades III, C. Oliver, and R. Zhang. Lfsc for smt proofs: Work in progress. 878:21–27, 01 2012.

[45] G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983.

[46] Mark van Atten. The Development of Intuitionistic Logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy.* Metaphysics Research Lab, Stanford University, Summer 2022 edition, 2022.

Appendix A

Complete list of theorems and tactics

Note: throughout the rules, assume that `p`, `q` and `r` are ranging over `Prop`, `l` over `List Prop`, `perm` over `List Nat` and `a` and `b` are ranging over either `Int` or `Rat`

Name	Statement	Implementation
`notImplies1`	`¬ (p → q) → p`	theorem
`notImplies2`	`¬ (p → q) → ¬ q`	theorem
`equivElim1`	`p = q → ¬ p ∨ q`	theorem
`equivElim2`	`p = q → p ∨ ¬ q`	theorem
`notEquivElim1`	`¬ (p = q) → p ∨ q`	theorem
`notEquivElim2`	`¬ (p = q) → ¬ p ∨ ¬ q`	theorem
`iteElim1`	`ite p q r → ¬ p ∨ q`	theorem
`iteElim2`	`ite p q r → p ∨ r`	theorem
`notIteElim1`	`¬ ite p q r → ¬ p ∨ ¬ q`	theorem
`notIteElim2`	`¬ ite p q r → p ∨ ¬ r`	theorem
`contradiction`	`p → ¬ p → False`	theorem
`orComm`	`p ∨ q → q ∨ p`	theorem
`orAssoc`	`p ∨ q ∨ r → (p ∨ q) ∨ r`	theorem
`orAssocConv`	`p ∨ (q ∨ r) → p ∨ q ∨ r`	theorem
`congOrRight`	`(p → q) → p ∨ r → q ∨ r`	theorem
`congOrLeft`	`(p → q) → r ∨ p → r ∨ q`	theorem
`orImplies`	`(¬ p → q) → p ∨ q`	theorem
`orImplies`$_2$	`¬ p ∨ q → (p → q)`	theorem
`orImplies`$_3$	`p ∨ q → (¬ p → q)`	theorem

`scope`	`(p → q) → ¬ p ∨ q`	theorem
`impliesElim`	`(p → q) → ¬ p ∨ q`	theorem
`deMorganSmall`	`¬ (p ∨ q) → ¬ p ∧ ¬ q`	theorem
`deMorganSmall`$_2$	`¬ p ∧ ¬ q → ¬ (p ∨ q)`	theorem
`deMorganSmall`$_3$	`¬ (p ∧ q) → ¬ p ∨ ¬ q`	theorem
`notNotElim`	`¬ (¬ p) → p`	theorem
`notNotIntro`	`p → ¬ (¬ p)`	theorem
`deMorgan`	`¬ orN (notList l) → andN l`	theorem
`deMorgan`$_2$	`andN l → ¬ orN (notList l)`	theorem
`deMorgan`$_3$	`¬ orN l → andN (notList l)`	theorem
`cnfAndNeg`	`andN l ∨ orN (notList l)`	theorem
`cnfAndPos`	`¬ (andN l) ∨ l.getD i True`	theorem
`cnfOrNeg`	`orN l ∨ ¬ l.getD i False`	theorem
`cnfOrPos`	`¬ (orN l) ∨ orN l`	theorem
`cnfImpliesPos`	`¬ (p → q) ∨ ¬ p ∨ ¬ q`	theorem
`cnfImpliesNeg1`	`(p → q) ∨ p`	theorem
`cnfImpliesNeg2`	`(p → q) ∨ ¬ q`	theorem
`cnfEquivPos1`	`¬ (p = q) ∨ ¬ p ∨ q`	theorem
`cnfEquivPos2`	`¬ (p = q) ∨ p ∨ ¬ q`	theorem
`cnfEquivNeg1`	`(p = q) ∨ p ∨ q`	theorem
`cnfEquivNeg2`	`(p = q) ∨ ¬ p ∨ ¬ q`	theorem
`cnfItePos1`	`¬ (ite p q r) ∨ ¬ p ∨ q`	theorem
`cnfItePos2`	`¬ (ite p q r) ∨ p ∨ r`	theorem
`cnfItePos3`	`¬ (ite p q r) ∨ q ∨ r`	theorem
`cnfIteNeg1`	`(ite p q r) ∨ ¬ p ∨ ¬ q`	theorem
`cnfIteNeg2`	`(ite p q r) ∨ p ∨ ¬ r`	theorem
`cnfIteNeg3`	`(ite p q r) ∨ ¬ q ∨ ¬ r`	theorem

`iteIntro`	`ite p ((ite p q r) = q)`
	`((ite p q r) = r)`
		theorem

eqResolve	p → p = q → q	theorem
dupOr	p ∨ p ∨ q → p ∨ q	theorem
dupOr$_2$	p ∨ p → p	theorem
andElim	andN l → l.getD i True	tactic
notOrElim	¬ (orN l) → ¬ (l.getD i False)	tactic
notAnd	¬ (andN l) → orN (notList l)	theorem
modusPonens	p → (p → q) → q	theorem
trueIntro	p → p = True	theorem
trueElim	p = True → p	theorem
trueElim$_2$	True = p → p	theorem
falseIntro	¬ p → p = False	theorem
falseIntro$_2$	¬ p → False = p	theorem
falseElim	p = False → ¬ p	theorem
falseElim$_2$	False = p → ¬ p	theorem
negSymm	¬ (p = q) → ¬ (q = p)	theorem
factor	orN l → orN (removeDuplicates l)	tactic
liftOrNToImp	orN l → andN (take i (notList l)) → orN (drop i l)	tactic
liftOrNToNeg	orN l → ¬ (andN l)	tactic
permutateClause	orN l → orN (permutate perm l)	tactic
pull	orN l → l.getD i False ∨ orN (eraseIdx i l)	tactic
R1	orN l$_1$ → orN l$_2$ → orN ((erase p l$_1$) ++ (erase (¬ p) l$_2$))	tactic
R2	orN l$_1$ → orN l$_2$ → orN ((erase (¬ p) l$_1$) ++ (erase p l$_2$))	tactic
arithMulPos	m > 0 ∧ a < b → m * a < m * b	tactic
arithMulNeg	m < 0 ∧ a < b → m * a > m * b	tactic
tightLb	a < b → a ≤ (ceil b) - 1	tactic
tightUb	a > b → a ≥ (floor b) + 1	tactic

`trichotomy`	`¬ (a < b) → ¬ (a = b) → a > b`	tactic