LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

CHAIR OF THEORETICAL COMPUTER SCIENCE AND THEOREM PROVING

Proof Representation in `cvc5`

Özge Özenoglu

Bachelor's Thesis
in Computer Science plus Statistics

Supervisor: Prof. Dr. Jasmin Blanchette

Advisor: Lydia Kondylidou

Submission Date: April 24, 2025

Disclaimer

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, April 24, 2025 Özge Özenoglu

Acknowledgments

First and foremost, I would like to thank my advisor, Lydia Kondylidou, for her guidance during this thesis project and for sparking my interest in SMT solving. I am also grateful to Prof. Dr. Blanchette for offering this thesis topic and for providing valuable feedback during my presentation. Finally, I would like to thank my family and Caner for their emotional support and for being my source of motivation everyday.

Abstract

Satisfiability Modulo Theories (SMT) solvers, such as `cvc5`, handle a variety of theories for deciding the satisfiability of first-order logical formulas. Their efficiency and flexibility have made them central in fields like program verification, testing, and model checking. However, the proofs generated for unsatisfiable results are often very long and the reasoning is not easy to follow. This thesis introduces a decoder that converts `cvc5`'s default proof format into a clearer and more user-friendly version while preserving the logical correctness. The decoder is built in a modular way, with separate steps for refactoring the proof, parsing it into its components, simplifying the logical expressions, and finally formatting the output. This approach not only makes the decoding process transparent but also allows for easy future extensions. Tests with proof files from different theories showed that the decoder currently works best for small to medium-length proofs that are not too deeply nested. Although the current implementation only supports a subset of `cvc5`'s theories, future work will focus on extending its coverage, adding theory-specific simplifications, and even handling higher-order logic. Overall, this decoder sets the groundwork for creating more accessible and understandable proofs in automated reasoning tools.

Keywords: Satisfiability-Modulo Theories, cvc5, First-Order Logic, SMT-LIB.

Contents

1 Introduction

A major research area in computer science addresses with software verification and methods to prove the correctness of software systems. A cornerstone in this area is logic, which enables a precise and formal description of the system's properties and requirements. When these properties are expressed as logical formulas, automated reasoning tools can be used to detect errors or to prove compliance with specified requirements. However, evaluating more complex system properties often requires adding specialized logical theories that extend beyond mere Boolean logic. Building on these foundations, advanced reasoning tools have been developed to decide the satisfiability of logical formulas - that is, to determine whether a formula can be made true or not - which forms the basis of Satisfiability Modulo Theories (SMT) solvers.

SMT solvers are powerful reasoning tools for deciding the satisfiability of logical formulas based on various theories, such as linear arithmetic, arrays, bit vectors, and strings (Barrett et al. 2009). At the core of an SMT solver, a Boolean satisfiability (SAT) solver transforms the original problem into a purely Boolean representation. The SAT solver then determines whether there is a model that makes the logical formula true. If so, the SMT solver checks whether the candidate solution is consistent with the underlying theory. If consistency is confirmed, the SMT solver returns the satisfying assignment of variables. Otherwise, it proves that no such assignment exists.

Because of their efficiency in solving complex problems, SMT solvers are widely applied in both industry and academia. Fields of application include program verification (Fan et al. 2022; De Moura & Bjørner 2011; Leino 2010), testing (Xue et al. 2024; Felbinger & Schwarzl 2014; Chimisliu & Wotawa 2012), model checking (Farias et al. 2024; Lee et al. 2022; Pereira et al. 2016), program synthesis (Wang et al. 2024; Itzhaky et al. 2021), security (ter Beek et al. 2024; Kulik et al. 2022; Godefroid et al. 2012) and scheduling (De Moura & Bjørner 2011).

Today, there are many SMT solvers available. State-of-the-art solvers are `Z3` (de Moura & Bjørner 2008), `veriT` (Bouton et al. 2009), `SMTInterpol` (Christ et al. 2012), `MathSAT5` (Cimatti et al. 2013), `Yices 2` (Dutertre 2014), `Boolector 3.0` (Niemetz et al. 2018), and `Bitwuzla` (Niemetz & Preiner 2020).

A slightly different approach is taken by the cooperating validity checker (CVC) solvers. Unlike the previously mentioned solvers, CVC solvers combine independent modules for specific theories (such as arrays or linear arithmetic) using a Nelson-Oppen-style architecture (Stump et al. 2002). The latest solver in this family, `cvc5` (Barbosa et al. 2022), is already used in industry. For instance, Amazon's AWS uses `cvc5` in their identity and access management systems to verify access control policies (ter Beek et al. 2024).The `cvc5` solver combines the benefits of the cooperative approach with improved algorithms and extended support for theories (Barbosa et al. 2022).

For unsatisfiable (unsat) problems, `cvc5` produces a proof in a special proof calculus. Though proof rules are well-documented, the proof generated is often long, and strongly convoluted, making it difficult and time-consuming for the user to follow the underlying reasoning and logical conclusions. To address this issue, the present work introduces a "decoder" which aims to build the groundwork for transforming `cvc5` proofs in its default format, namely LFSC[1], into a more human-readable and traceable format. For example, an expression such as `(not (forall (A) (=> (A) (B))))`, can be translated by the decoder into the logically equivalent but easier-to-follow

[1]The LFSC proof format is based on the LF logical framework extended with computational side conditions, see `https://cvc5.github.io/docs/cvc5-1.0.0/proofs/output_lfsc.html`.

expression `(exists(A)(and(A)(not(B))))`. Such simplifications helps users to better understand the reasoning process, facilitates verification and debugging, and ultimately may strengthen trust in automated reasoning tools.

The present work is structured as follows. Section 2 provides an overview of the theoretical background and applications of SAT and SMT solvers. Moreover, it illustrates the internal structure of SMT solvers through a concrete example and introduces the `cvc5` solver. Section 3 describes the general structure of `cvc5` proofs and presents the theoretical foundations of the decoder. Section 4 details the implementation of the decoder, highlighting its primary modules and data structures. Section 5 evaluates the tool and outlines the theories currently supported by the decoder. Finally, Section 6 discusses future directions, followed by a conclusion of the project in Section 7.

2 Background

Logic is an essential cornerstone of computer science and plays an important role in the formal specification and verification of systems. Various logical frameworks exist with differing expressive power. Propositional logic deals with simple, atomic propositions and their combinations. For example, consider the statement “Humans are mortal”. In propositional logic, this can be represented by letting A denote “human” and B denote “mortal.” The implication $A \to B$ represents the statement that if something is human, then it is mortal. First-order logic (FOL), also known as predicate logic, is more expressive because it allows quantified statements about objects and their properties. For instance, we can formalize statements like “Every prime number larger than two is odd" as $\forall x.\ ((\text{prime}(x) \land x > 2) \to \text{odd}(x))$. This universal quantification is beyond the scope of propositional logic. Higher-order logic (HOL) further extends the expressiveness by permitting quantification over predicates and functions. For example “There exists a property P that holds exactly when numbers are smaller than ten" can be expressed as $\exists P.\ \forall n \in \mathbb{N}.\ (P(n) \leftrightarrow (n < 10))$. Here, P is a predicate variable representing a property (e.g., being smaller than ten) shared by all numbers.

SMT solvers use these logical frameworks to algorithmically decide the satisfiability of formulas across various theories. This section describes the theoretical foundations and internal architecture of SAT and SMT solvers, explaining their functionality through a concrete example problem from linear integer arithmetic. Furthermore, this section introduces `cvc5`, a high-performance and efficient SMT solver.

2.1 Satisfiability Modulo Theories

The Boolean satisfiability problem (SAT) asks whether there is an assignment of variables such that the propositional formula is satisfied, i.e., evaluates to true. This seemingly inconspicuous problem definition is of fundamental importance in complexity theory. In fact, the SAT problem has been proven to be NP-complete (Cook 1971; Levin 1973). This constitutes that it is the hardest problem in NP and that there likely[2] is no efficient algorithm that decides SAT.

Satisfiability Modulo Theories (SMT) problems extend the classic SAT problem by additionally allowing first-order formulas. The term "theories" refers to domain-specific knowledge (such as

[2]Not least until it is proven that $P = NP$, which is one of the unsolved seven millennial math problems.

linear arithmetic, arrays, bit vectors) expressed in first-order logic. As SMT problems are a generalization of SAT problems, and SAT is NP-complete, it can be deduced that SMT is generally NP-hard. As such, there is also no known algorithm that efficiently solves all SMT problems. An SMT solver is a tool that decides SMT problems. Despite being known for computationally intensive and challenging, interestingly, state-of-the-art-solvers are in fact able to decide many real-world problems in reasonably good time (Barrett et al. 2014).

An SMT solver is typically divided into two main components (De Moura & Bjørner 2011): a propositional core (SAT solver) and specialized theory solvers (see Figure 1). Two major frameworks for deciding the satisfiability with respect to theory T are the Davis-Putnam-Logemann-Loveland (DPLL(T)) algorithm (Nieuwenhuis et al. 2006) and Conflict-Driven Clause Learning (CDCL(T)) (see Bonacina (2018)). To illustrate the processes, consider the following problem from linear integer arithmetic[3]:

$$\exists x \in \mathbb{Z} : (x \leq 0 \vee x \geq 1) \wedge (x \geq 2 \vee x \leq 0). \tag{1}$$

To enable the SAT solver to process the formula, it is first abstracted into Boolean logic. Note that if the original formula is not already expressed in Conjunctive Normal Form[4] (CNF), it is converted into that form first. As the formula in Equation 1 is in CNF, this step is skipped and the atoms are abstracted as $p \equiv x \leq 0$, $q \equiv x \geq 1$, $r \equiv x \geq 2$. Thus, the formula becomes:

$$(p \vee q) \wedge (r \vee p). \tag{2}$$

The DPLL algorithm (Davis & Putnam 1960; Davis et al. 1962) is a branch and backtracking algorithm that gradually simplifies the formula by assigning truth values to the literals following rules unit propagation[5] and pure literal elimination[6]. Suppose the SAT solver initially assigns $p \mapsto$ `T` and $r \mapsto$ `T`. This corresponds to the theory assignment:

$$\{x \leq 0, x \geq 2\}. \tag{3}$$

The linear integer arithmetic theory immediately recognizes that there exists no integer x that simultaneously satisfies $x \leq 0$ and $x \geq 2$ so that a theory conflict is detected. After backtracking, the SAT solver selects a different Boolean assignment. For instance, it might now choose $q \mapsto T$ and $r \mapsto T$, leading to the theory assignment $\{x \geq 1, x \geq 2\}$. This simplifies to $x \geq 2$, which is a valid solution in light of the theory.

CDCL(T) extends the DPLL(T) framework in that it not only backtracks when a conflict is encountered (such as in Equation 3) but also analyzes the conflict to learn a new clause that prevents the same combination of decisions from being repeated (Bonacina 2018). In a CDCL(T) framework, the solver would add a new clause $\neg p \vee \neg r$ to express that not both $x \leq 0$ and $x \geq 2$ can be true. With the new conflict clause added, the solver backtracks not necessarily step-by-step but directly to

[3]Note that the input problem must first adhere to the syntactical requirements of the solver. In order to have a unified syntax, the standardized language SMT-LIB has been developed (Barrett et al. 2016). Version 2 of the standard is nowadays supported by all modern SMT solvers. For simplicity a mathematical representation of the problem is shown here.

[4]A propositional formula ϕ is in CNF iff $\phi = \bigwedge_{i=1}^{n} \left(\bigvee_{j=1}^{m_i} l_{i,j} \right)$, where $n, m_i \in \mathbb{N}$ and $l_{i,j}$ is a literal.

[5]Each clause with a single unassigned literal must take the value that fulfills the clause.

[6]Assign literals that appear only in one polarity throughout the formula.

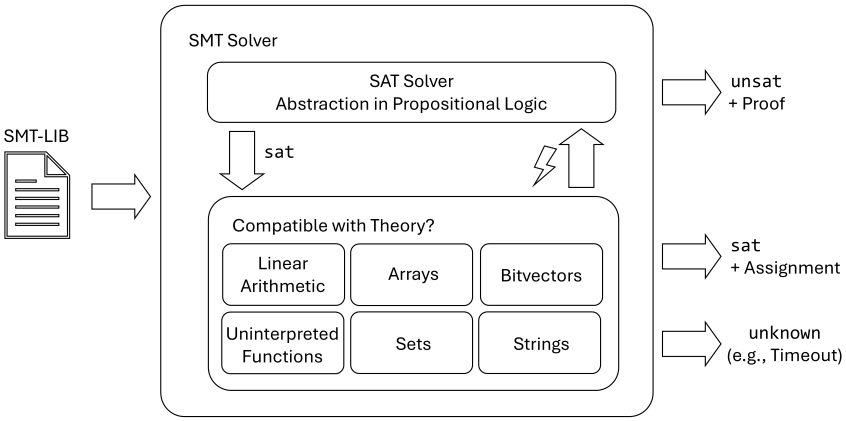

Figure 1: Abstract structure of an SMT solver. For an SMT problem expressed in a special input format (typically SMT-LIB), the solver checks whether the problem is satisfiable. At the core is a SAT solver that decides the Boolean representation of the problem. If a fulfilling model is found that is also compatible with the specified theorie(s), the SMT solver returns `sat` and the assignments. If no such assignment can be found, the solver return `unsat` and a proof. Some problems cannot be decided (e.g., because they take too long), in which case the solver outputs `unknown`.

the decision level where the conflict can be resolved (Bonacina 2018). As the new clause forces the solver to avoid the conflicting assignment, it might now choose $q \mapsto T$ and $r \mapsto T$. The resulting theory assignment is $\{x \geq 1, x \geq 2\}$, which is consistent with the theory. The SMT solver returns `sat` (satisfiable) and an assignment of literals that satisfy the formula (e.g., $x \mapsto 2$).

2.2 `cvc5`

The `cvc5` SMT solver is the most recent solver for first-order and higher-order logic in the CVC series. It builds on its predecessor `CVC4` (Barrett et al. 2011) by implementing numerous enhancements and optimizations (Barbosa et al. 2022).

The architecture of `cvc5` is highly modular. The SMT solver is subdivided into the preprocessor, rewriter, the propositional engine and the theory engine (see also Fig. 1 of (Barbosa et al. 2022)). The core of the propositional engine builds the modified Minisat SAT solver (Eén & Sörensson 2004). In contrast to the DPLL(T)-based `CVC4`, `cvc5` uses the CDCL(T) algorithm as the central solution mechanism.

`cvc5` is an SMT solver that offers extensive support for a wide range of theories. In addition to all standardized SMT-LIB theories, cvc5 also covers several non-standard ones. It supports fundamental theories such as linear arithmetic (for both integer and real numbers), non-linear arithmetic, arrays, bit-vectors, and algebraic datatypes. Moreover, cvc5 provides support for floating-point arithmetic, sets and relations, separation logic, strings and sequences, as well as uninterpreted functions and quantifiers.

Regarding its interfaces, `cvc5` supports the SMT-LIB (SMT-LIBv2) format as its standard input, while also accepting languages like SyGuS and TPTP. In fact, it includes an internal parser that converts TPTP problems into SMT-LIB format. For unsatisfiable problems, `cvc5` can produce proofs in several formats, such as LFSC, Alethe, DOT, CPCP, and Lean 4, with the default proof format being LFSC (Logical Framework with Side Conditions). In this format, a proof is structured as a sequence of inference steps, each explicitly annotated with an inference rule.

While the LFSC-based proof format is highly structured and formal, it is often challenging to read and interpret for human readers. The proofs can be long, nested, and contain auxiliary derivation steps that complicate the understanding of the main reasoning. The primary objective of this thesis is to improve the comprehensibility and interpretability of `cvc5` proofs by developing a “decoder" that transforms LFSC proofs into a human-readable, traceable format while preserving the underlying logical structure. In the following section, the detailed structure of `cvc5` proofs will be described, laying the foundation for the subsequent decoding process.

3 Proof Representation

The following section describes how `cvc5` proofs are decoded in four main steps. First, the generated proof is refactored to remove unnecessary details (see Section 3.2). Next, the structure of the proof is utilized to extract components from the proof (see Section 3.3). This is followed by a lexical analysis of the component containing the logical expression (see Section 3.4). Finally, the expression is parsed into an abstract syntax tree and repeatedly simplified (see Section 3.5).

3.1 Running Example

To illustrate the decoding process, consider the following minimal unsatisfiable problem from set theory. It introduces a binary predicate, $subset(A,B)$, which denotes that set A is a subset of set B:

$$\neg(\forall A : (subset(A,A) \Rightarrow subset(A,A))) \qquad \text{(Axiom 1)}$$
$$\forall A : subset(A,A) \qquad \text{(Axiom 2)}$$

Axiom 1 states that not all sets are subsets of themselves, whereas Axiom 2 asserts that A is a subset of itself (which is always true, given reflexivity of the subset relation). The contradiction and hence the unsatisfiability arises because Axiom 1 is equivalent to claiming that there exists at least one set for which the self-implication fails, which is impossible in classical logic. As a result, no model can satisfy the problem. The problem definition in SMT-LIB syntax is given in Listing 1.

Listing 1: SMT-LIB representation of the set problem.

```
(declare-sort $$unsorted 0)
(declare-fun tptp.subset ($$unsorted $$unsorted) Bool)
(assert (not (forall ((A $$unsorted)) (let ((_let_1 (tptp.subset A A)))
        (=> _let_1 _let_1)))))
(assert (forall ((A $$unsorted)) (tptp.subset A A)))
(check-sat)
```

Upon processing the SMT-LIB formatted problem, the `cvc5` solver applies its internal proof mechanisms to determine the satisfiability of a given problem. For this example, the solver concludes that it is unsatisfiable. Moreover, the solver generates the detailed proof in Listing 2 that outlines its reasoning steps. In the following sections, proof line `(assume @p1 (not @t4))` is used to showcase the functioning of the decoder.

Listing 2: Proof production of the set problem.

```
unsat
(declare-type $$unsorted ())
(declare-const tptp.subset (-> $$unsorted $$unsorted Bool))
(define @t1 () (eo::var "A" $$unsorted))
(define @t2 () (tptp.subset @t1 @t1))
(define @t3 () (@list @t1))
(define @t4 () (forall @t3 (=> @t2 @t2)))
(define @t5 () (forall @t3 true))
(assume @p1 (not @t4))
(assume @p2 (forall @t3 @t2))
; WARNING: add trust step for TRUST_THEORY_REWRITE
; trust TRUST_THEORY_REWRITE
(step @p3 :rule trust :premises () :args ((= (not true) false)))
; trust TRUST_THEORY_REWRITE
(step @p4 :rule trust :premises () :args ((= @t5 true)))
; trust TRUST_THEORY_REWRITE
(step @p5 :rule trust :premises () :args ((= @t4 @t5)))
(step @p6 :rule trans :premises (@p5 @p4))
(step @p7 :rule cong :premises (@p6) :args (not))
(step @p8 :rule trans :premises (@p7 @p3))
(step @p9 false :rule eq_resolve :premises (@p1 @p8))
```

3.2 Refactoring

In the first step of decoding, the proof is refactored to ensure it is ready for parsing. This involves removing unnecessary details such as comments (marked by `;`) or special characters (e.g., `$`). Additionally, names of type variables are made more intuitive and replaced throughout the proof. For instance, in the running example, the type `unsorted` is replaced with the free variable name `a`, ensuring the cleaner proof shown in Listing 3.

Listing 3: Refactored version of the proof for the set problem.

```
unsat
(declare-type a ())
(declare-const subset (-> a a Bool))
(define @t1 () (eo::var "A" a))
(define @t2 () (subset @t1 @t1))
(define @t3 () (@t1))
(define @t4 () (forall @t3 (=> @t2 @t2)))
(define @t5 () (forall @t3 true))
(assume @p1 (not @t4))
(assume @p2 (forall @t3 @t2))
(step @p3 :rule trust :premises () :args ((= (not true) false)))
(step @p4 :rule trust :premises () :args ((= @t5 true)))
(step @p5 :rule trust :premises () :args ((= @t4 @t5)))
(step @p6 :rule trans :premises (@p5 @p4))
(step @p7 :rule cong :premises (@p6) :args (not))
(step @p8 :rule trans :premises (@p7 @p3))
(step @p9 false :rule eq_resolve :premises (@p1 @p8))
```

3.3 cvc5 Proof Structure

The proofs generated by cvc5 are well structured, which is used in the following to parse them generically. Each proof line thereby corresponds to one inference step. The main building blocks of each proof line are reduced to the following components[7]:

<TYPE><TAG><BODY$_{\texttt{TYPE}}$>

<TYPE>	Description
declare-type	Introduces a new type into the proof. Component <BODY$_{\texttt{declare-type}}$> is of the form <TYPE><NOTE>.
declare-const	Introduces a new constant into the proof. It specifies that a certain identifier exists (e.g., a function declaration) and belongs to a certain type. Component <BODY$_{\texttt{declare-const}}$> is of the form <ARGS><NOTE>.
define	Introduces a definition of expressions or variables that are used in the subsequent proof. Component <BODY$_{\texttt{define}}$> is of the form <PREMS><ARGS>.
assume	Denotes an assumption. A formula is assumed to be true, so that it can be used for further inference steps in the course of proof. Component <BODY$_{\texttt{assume}}$> is of the form <ARGS>.
step	Summarizes a single derivation step. It documents which previously derived partial results were used (premises) and which inference (e.g. logical transformation) led to the new conclusion (rule). Component <BODY$_{\texttt{step}}$> is of the form <RULE><PREMS><ARGS>.

Table 1: Overview of the Decoder's supported instances of the <TYPE> component.

The <TYPE> component denotes the specific derivation step, making the proof process more comprehensible. Table 1 provides an overview of the central types. The <TAG> component assigns a unique identifier (tag) to those proof lines that are cross-referenced later in the proof. A tag always begins with an @-symbol, followed by either a p or t and a number. Finally, the <BODY$_{\texttt{TYPE}}$> component defines the main part of the proof line and is divided into various subcomponents depending on the <TYPE>. A discription of the subcomponents that may form the body are provided in Table 2.

Although every component is important to later reproduce the proof without loss, the most important component is the <ARGS> component, since it contains the logical expression (in prefix notation). At this stage, however, the logical expression may still contain cross-references that must first be resolved with the argument of the referenced line prior to the lexical analysis in Section 3.4. Formally, the cross-references are resolved sequentially. For example, the argument component

[7] See also the documentation at https://cvc5.github.io/docs/cvc5-1.2.1/proofs/output_cpc.html.

Component	Description
<NOTE>	Additional information, such as function signatures, that should only occur once in the decoded proof.
<ARGS>	A logical expression in prefix notation possibly containing cross-references (tags) to previous proof lines.
<PREMS>	Cross-references (tags) to previous proof-lines.
<RULE>	The proof rule applied in the inference step. Common rules include `trust`, `trans`, `cong`. Proof rules are detailed in the documentation at `https://cvc5.github.io/docs/cvc5-1.2.1/api/c/enums/cvc5proofrule.html#`.

Table 2: Components contained in <BODY>.

`(not @t4)` in proof line `(assume @p1 (not @t4))` is replaced step by step as follows:

```
(not @t4)                                                    (replace @t4)
(not (forall @t3 (=> @t2 @t2)))                              (replace @t2, @t3)
(not (forall (@t1) (=> (subset @t1 @t1) (subset @t1 @t1))))  (replace @t1)
(not (forall (A) (=> (subset (A) (A)) (subset (A) (A)))))
```

However, Section 4 implements a method that resolves the cross-references efficiently so that each line in itself already contains the fully resolved expression:

```
(not @t4)                                                    (replace @t4)
(not (forall (A) (or (not (subset (A) (A))) (subset (A) (A))))) (4)
```

The biggest challenge of decoding a given proof is to apply simplification rules without distorting the semantics of the logical formula. For this purpose, a robust mechanism is needed to (1) keep partial expressions grouped together (2) repeatedly apply simplification steps in order to completely simplify the formula and (3) flexibly handle not-coded lables (such as function names). Taking inspiration from compiler construction, an approach based on lexical analysis and parsing is used here to address these challenges (Thain 2020).

3.4 Lexical Analysis

Before the proof is simplified line by line, the logical formula is first scanned lexically. This process involves breaking each line into a stream of tokens, thereby isolating meaningful symbols, such as operators, quantifiers and literals, while filtering out irrelevant elements like whitespace. By tokenizing the formula, a version that is easier to work with is created for subsequent transformation steps. Since logical expressions are contained solely in the <ARGS> component, only this part is considered during the scanning (and parsing) step.

Let Σ be an alphabet including special characters and let $w \in \Sigma^*$ be a logical expression in a proof line. The tokenization procedure is modeled as follows. Let T denote the set of token types

that can be assigned to w, with the specific tokens determined by the tokenization rules R:

$$T = \{\texttt{LPAREN}, \texttt{RPAREN}, \texttt{NOT}, \texttt{AND}, \texttt{OR}, \texttt{IMP}, \texttt{EQ}, \texttt{FORALL}, \texttt{EXISTS}, \texttt{TRUE}, \texttt{FALSE}, \texttt{VAR}, \texttt{NUMBER}\}$$

$$R = \left\{ \begin{array}{rcl} \text{“(" } & \mapsto & \texttt{LPAREN}, \\ \text{“)" } & \mapsto & \texttt{RPAREN}, \\ \text{“not" } & \mapsto & \texttt{NOT}, \\ \text{“and" } & \mapsto & \texttt{AND}, \\ \text{“or" } & \mapsto & \texttt{OR}, \\ \text{“=>" } & \mapsto & \texttt{IMP}, \\ \text{“=" } & \mapsto & \texttt{EQ}, \\ \text{“forall" } & \mapsto & \texttt{FORALL}, \\ \text{“exists" } & \mapsto & \texttt{EXISTS}, \\ \text{“true" } & \mapsto & \texttt{TRUE}, \\ \text{“false" } & \mapsto & \texttt{FALSE}, \\ \text{[a-zA-Z][a-zA-Z0-9\_]}^* & \mapsto & \texttt{VAR}, \\ \text{“-"?[0-9]+} & \mapsto & \texttt{NUMBER}, \\ \text{(“-" | “+" | “/" | “*" | “>=" | “<=" | “>" | “<")} & \mapsto & \texttt{VAR}, \\ \text{[\textbackslash t\textbackslash n]+} & \mapsto & \varepsilon, \end{array} \right\}$$

Additionally, a catch-all rule is defined which captures any single character from Σ that is not matched by the above rules, ensuring that every part of the input is tokenized. Using this set of token rules, we can formally define $f : \Sigma^* \to (T \times \Sigma^*)^*$ as a recursive function that divides an input string into a sequence of tokens:

$$f(w) = \begin{cases} \varepsilon, & \text{for } w = \varepsilon, \\ (t,s) \circ f(w'), & \text{for } w = s \cdot w', \text{where } s \text{ is the longest prefix of } w \\ & \text{with } (P \mapsto t) \in R \text{ and } t \neq \varepsilon, \\ f(w'), & \text{else} \end{cases},$$

where ε is the empty sequence (i.e, whitespace) and $\circ$ is the concatenation of token sequences. This definition states that for a non-empty input string, its longest prefix s that belongs to a token t is searched for first, which is the so-called maximal-munch tokenization (Reps 1998). If t is not a whitespace token, the pair (t,s) is added to the resulting token sequence. For example, expression $w =$ `(not (forall (A) (or (not (subset (A) (A))) (subset (A) (A)))))` from Section 3.3 is tokenized as follows:

$$f(w) = \begin{bmatrix} \texttt{(LPAREN,``(")} \\ \texttt{(NOT,``not")} \\ \texttt{(LPAREN,``(")} \\ \texttt{(FORALL,``forall")} \\ \texttt{(LPAREN,``(")} \\ \texttt{(VAR,``A")} \\ \texttt{(RPAREN,``)")} \\ \texttt{(LPAREN,``(")} \\ \texttt{(OR,``or")} \\ \texttt{(LPAREN,``(")} \\ \texttt{(NOT,``not")} \\ \texttt{(LPAREN,``(")} \\ \texttt{(VAR,``subset")} \\ \texttt{(LPAREN,``(")} \\ \texttt{(VAR,``A")} \\ \texttt{(RPAREN,``)")} \\ \texttt{(LPAREN,``(")} \\ \texttt{(VAR,``A")} \\ \texttt{(RPAREN,``)")} \\ \texttt{(RPAREN,``)")} \\ \texttt{(LPAREN,``(")} \\ \texttt{(VAR,``subset")} \\ \texttt{(LPAREN,``(")} \\ \texttt{(VAR,``A")} \\ \texttt{(RPAREN,``)")} \\ \texttt{(LPAREN,``(")} \\ \texttt{(VAR,``A")} \\ \texttt{(RPAREN,``)")} \\ \texttt{(RPAREN,``)")} \\ \texttt{(RPAREN,``)")} \\ \texttt{(RPAREN,``)")} \\ \texttt{(RPAREN,``)")} \end{bmatrix}$$

3.5 Parser

To accurately capture the grouping and nesting of the logical expression, the tokens are converted into an abstract syntax tree (AST), with sub-expressions organized into independent sub-trees (see Section 3.5.1). This approach allows simplification rules to be applied precisely without altering the overall meaning of the expression (see Section 3.5.2). The parser is based on the following context-free grammar[8] G representing the semantics of a first-order logical formula in a `cvc5` proof line:

$$\begin{aligned} G &= (V, \Sigma, P, S) \\ V &= \{\text{input},\ \text{expr},\ \text{arglist}\} \\ \Sigma &= \{\texttt{LPAREN},\texttt{RPAREN},\texttt{NOT},\texttt{AND},\texttt{OR},\texttt{IMP},\texttt{EQ},\texttt{FORALL},\texttt{EXISTS},\texttt{TRUE},\texttt{FALSE},\texttt{VAR},\texttt{NUMBER}\} \\ S &= \text{input} \end{aligned}$$

[8] A grammar (V,Σ,P,S) is context-free iff for all $(l \to r) \in P : l \in V$.

$P =$ input $\rightarrow$ expr

expr $\rightarrow$ VAR
| NUMBER
| TRUE
| FALSE
| LPAREN expr RPAREN
| LPAREN exprlist RPAREN
| LPAREN NOT expr RPAREN
| LPAREN AND expr expr RPAREN
| LPAREN OR expr expr RPAREN
| LPAREN IMP expr expr RPAREN
| LPAREN EQ expr expr RPAREN
| LPAREN FORALL LPAREN VAR RPAREN expr RPAREN
| LPAREN EXISTS LPAREN VAR RPAREN expr RPAREN
| LPAREN VAR RPAREN
| LPAREN VAR arglist RPAREN

arglist $\rightarrow$ expr
| expr arglist

exprlist $\rightarrow$ expr
| expr exprlist

3.5.1 Abstract Syntax Tree

An abstract syntax tree (AST) is a hierarchical structure that is used in compiler construction to represent the essential structure of a program while omitting extraneous syntactic details (Thain 2020). We will use the AST here to capture the structure of logical expressions.

During parsing, the sequence of tokens from the respective proof line is matched against the grammar G. Specifically, the parser sequentially compares (and possibly matches) each token to a production rule in G. The AST is then constructed recursively. For every production rule in G, a corresponding semantic action is executed to create an AST node. In this process, five categories of actions are distinguished: atomic expressions, groupings, operators, quantifiers, and function applications. These fundamental components enable the recursive parsing of complex expressions, resulting in a deeply nested tree structure. To illustrate this, Figure 8 shows the AST corresponding to the token sequence $f(w)$ of Expression (4) from Section 3.4.

Atomic Expressions

If an atomic expression $\phi \in \{\texttt{VAR}, \texttt{NUMBER}, \texttt{TRUE}, \texttt{FALSE}\}$ is recognized, an AST node $\texttt{NODE}_\phi$ is instantiated. This node is a leaf node, i.e, it has no children as they represent simple elements that cannot be broken down further. For example, expression $w = \texttt{A}$ is tokenized into $f(w) = [(\texttt{VAR}, \text{“}\texttt{A}\text{”})]$ and matches production rule VAR in G. This results in a leaf node $\texttt{NODE}_{\texttt{VAR}}$ with value A as shown in

Figure 2.

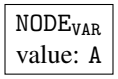

Figure 2: Graphical representation of an AST node for atomic expressions.

Groupings

If a token sequence is recognized by the production rule LPAREN expr RPAREN, a node of the type $\texttt{NODE}_{\texttt{GROUP}}$ is created. This node stores the parsed expression expr (the child) and thus preserves the explicit grouping through the brackets. For example, the node for the expression $w = (A)$ with $f(w) = [(\texttt{LPAREN}, \text{“(”}), (\texttt{VAR}, \text{“A”}), (\texttt{RPAREN}, \text{“)”})]$ is created as shown in Figure 3.

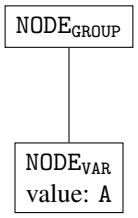

Figure 3: Graphical representation of an AST node for groupings.

If a token sequence is recognized by the production rule LPAREN exprlist RPAREN, however, the expressions inside the parentheses are first combined into a node $\texttt{NODE}_{\texttt{SEQ}}$ that preserves their sequential order before being encapsulated by the $\texttt{NODE}_{\texttt{GROUP}}$ node. For example, the expression $w = \texttt{(AB)}$ is tokenized and then parsed into an AST where the individual nodes for A and B are merged into a $\texttt{NODE}_{\texttt{SEQ}}$, which is then wrapped by a $\texttt{NODE}_{\texttt{GROUP}}$ node.

Operators

If a token sequence is recognized by the production rule capturing the unary operator not - that is, LPAREN NOT expr RPAREN - an AST node of type $\texttt{NODE}_{\texttt{NOT}}$ is created. Similarly, when a token sequence matches the production rule for binary operators - namely, LPAREN ϕ expr expr RPAREN with $\phi \in \{\texttt{AND}, \texttt{OR}, \texttt{IMP}, \texttt{EQ}\}$ - an AST node of type $\texttt{NODE}_{\phi}$ is instantiated. In the binary case, the node's left child holds the AST corresponding to the first operand, and its right child contains the AST corresponding to the second operand.

For example, consider the implication $w = \texttt{(=> (X) (Y))}$. Upon recognition of the token sequence $f(w)$, the parser constructs an AST node of type $\texttt{NODE}_{\texttt{IMP}}$, with the AST for X assigned as the left child and the AST for Y as the right child (see Figure 4).

Quantifiers

If a token sequence is recognized by the production rule LPAREN ϕ LPAREN VAR RPAREN expr RPAREN, where $\phi \in \{\texttt{FORALL}, \texttt{EXSIST}\}$ a node of the type $\texttt{NODE}_{\phi}$ is created. This node stores the parsed expression (i.e., the body of the quantifier) as its left child and the bound variable in its

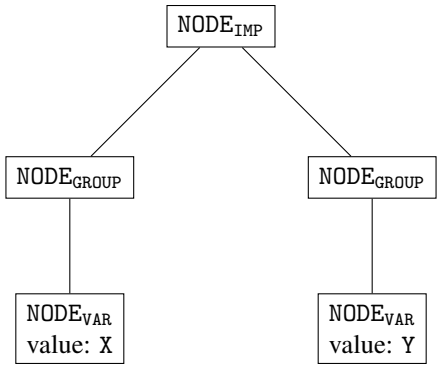

Figure 4: Graphical representation of an AST node for operators (implication).

value field, thereby capturing the semantics of the quantifier. For example, the tokenization $f(w)$ of expression $w =$ `(forall (X) Y)` is constructed to the AST in Figure 5.

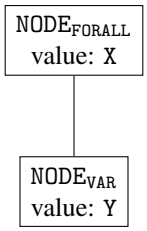

Figure 5: Graphical representation of an AST node for quantifiers (universal).

Function Applications

Function applications in `cvc5` proofs require careful modeling because they do not have a uniform name or a fixed number of arguments. They are modeled in the AST as nodes that denote the invocation of a function, with a clear distinction between applications with and without arguments.[9]

The production rule `LPAREN VAR RPAREN` represents a function application that does not supply any arguments. In this case, the function is captured by an AST node $\texttt{NODE}_{\texttt{FUN}}$ in which the function name is stored in the value field, while the absence of an argument list denotes that no further expressions are provided. For instance, Figure 6 shows the instantiated node for expression $w =$ `(fun)` and its tokenization $f(w)$.

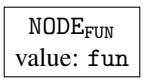

Figure 6: Graphical representation of an AST node for function applications without arguments.

Function applications with arguments are handled by the production rule `LPAREN VAR arglist RPAREN`. When such an expression is encountered, an AST node of type $\texttt{NODE}_{\texttt{FUN}}$ is created, with the function's name stored in its value field. Additionally, the argument list is processed

[9]This distinction is made out of necessity because proofs may sometimes contain expressions such as `(not)`.

recursively into a linked structure of $\texttt{NODE}_{\texttt{ARGLIST}}$ nodes, ensuring that the order of the arguments is preserved. For example, the AST to expression $w = (\texttt{funXYZ})$ and its tokenization $f(w) = [(\texttt{LPAREN}, \texttt{"("}), (\texttt{VAR}, \texttt{"fun"}), (\texttt{VAR}, \texttt{"X"}), (\texttt{VAR}, \texttt{"Y"}), (\texttt{VAR}, \texttt{"Z"}), (\texttt{RPAREN}, \texttt{")"})]$ is given in Figure 7.

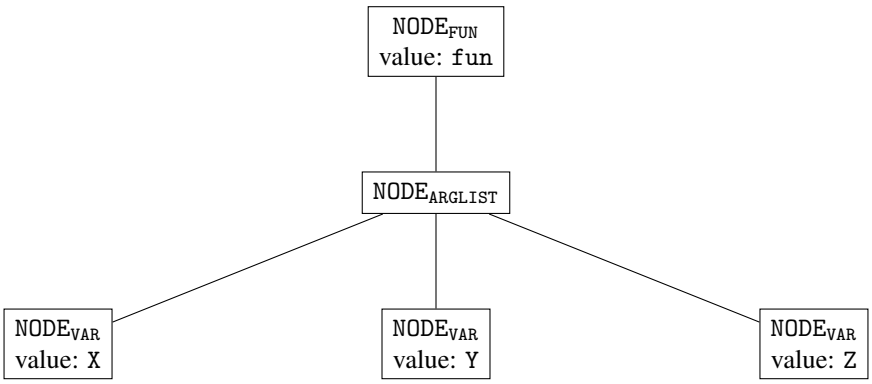

Figure 7: Graphical representation of an AST node for function applications with arguments.

3.5.2 Simplification

After constructing the AST for a logical expression, the parser recursively traverses the tree, incrementally applying simplification rules. In this process, it applies six specific rules for simplifying first-order logic expressions, continuing the simplification until no further reductions are possible. Let T be an abstract syntax tree with nodes $n \in T$. The function `simplifyAST` is defined recursively as:

$$\texttt{simplifyAST}(n) = \begin{cases} \varnothing, & \text{for } n = \varnothing \\ \texttt{applyRules}(n'), & \text{else} \end{cases},$$

where n' is the node with the recursively simplified sub-trees:

$$n' = (n.\text{left} \leftarrow \texttt{simplifyAST}(n.\text{left}),\ n.\text{right} \leftarrow \texttt{simplifyAST}(n.\text{right}))\,.$$

The function `applyRules` iteratively applies the following simplification rules to n' until no further changes can be made, meaning that the expression is maximally simplified:

$$\begin{aligned} \neg(\neg X) &\equiv X && \text{(Double Negation)} \\ \neg(X \wedge Y) &\equiv (\neg X) \vee (\neg Y) && \text{(De-Morgan)} \\ \neg(X \vee Y) &\equiv (\neg X) \wedge (\neg Y) && \text{(De-Morgan)} \\ X \Rightarrow Y &\equiv \neg X \vee Y && \text{(Implication)} \\ \neg(\forall\ (x)\ X) &\equiv \exists\ (x)\ (\neg X) && \text{(Negated Universal Quantifier)} \\ \neg(\exists\ (x)\ X) &\equiv \forall\ (x)\ (\neg X) && \text{(Negated Existential Quantifier)} \end{aligned}$$

For the running example, the AST of Expression (4) is fully simplified in three steps, namely simplifying the negated universal quantifier, applying De Morgan's law, and eliminating double

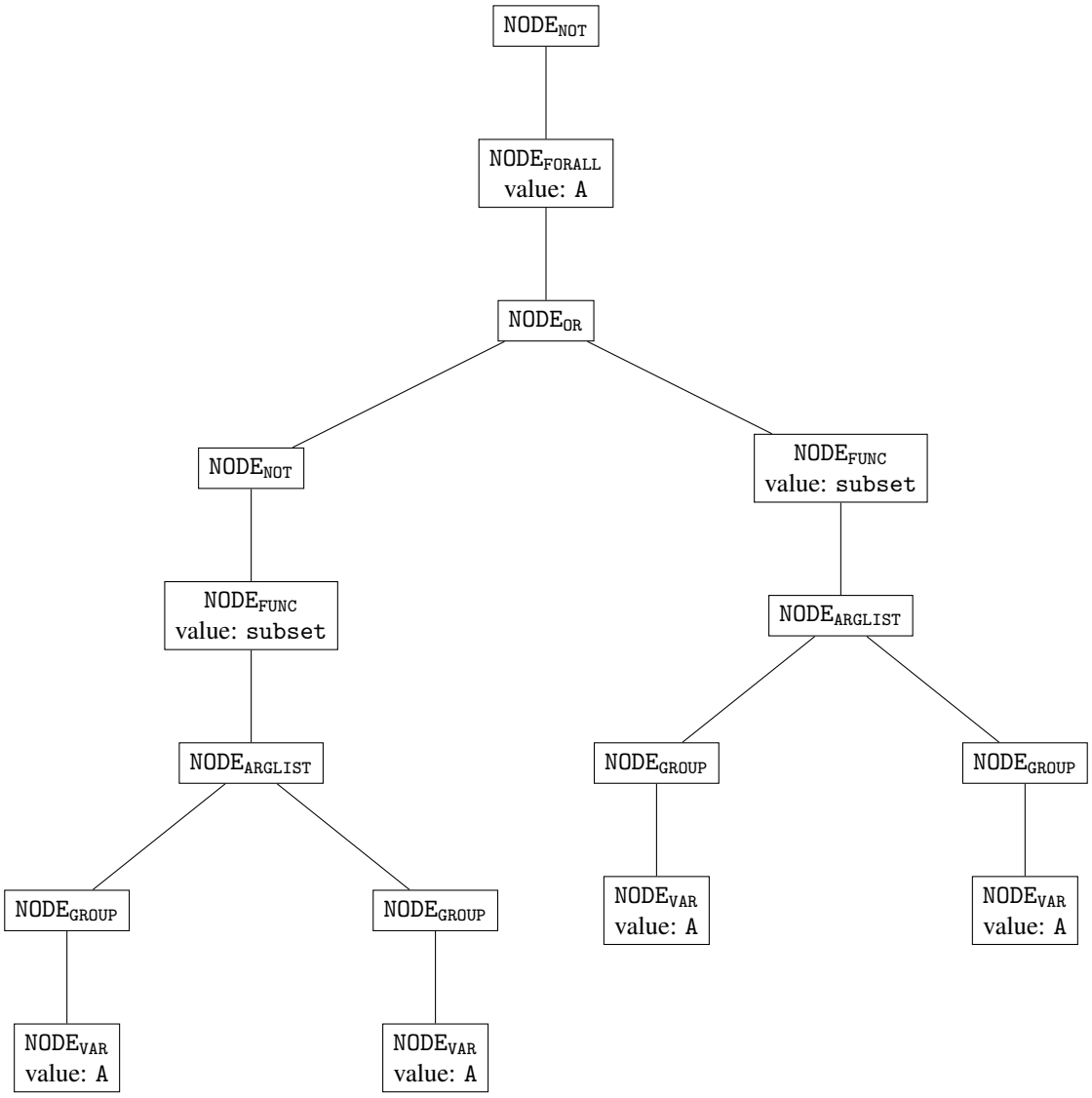

Figure 8: AST to the tokenization of Expression (4) (see Section 3.4).

negation. Figure 9 illustrates the resulting AST for the simplified expression.

$$\begin{aligned}\neg\Big(\forall A\ \big(\mathrm{or}\big(\neg(\mathrm{subset}(A,A)),\ \mathrm{subset}(A,A)\big)\big)\Big) &\Leftrightarrow \exists A\ \neg\Big(\mathrm{or}\big(\neg(\mathrm{subset}(A,A)),\ \mathrm{subset}(A,A)\big)\Big)\\ &\Leftrightarrow \exists A \neg\big(\neg(\mathrm{subset}(A,A))\big) \wedge \neg\big(\mathrm{subset}(A,A)\big)\\ &\Leftrightarrow \exists A\ \Big(\mathrm{subset}(A,A) \wedge \neg\big(\mathrm{subset}(A,A)\big)\Big)\end{aligned}$$

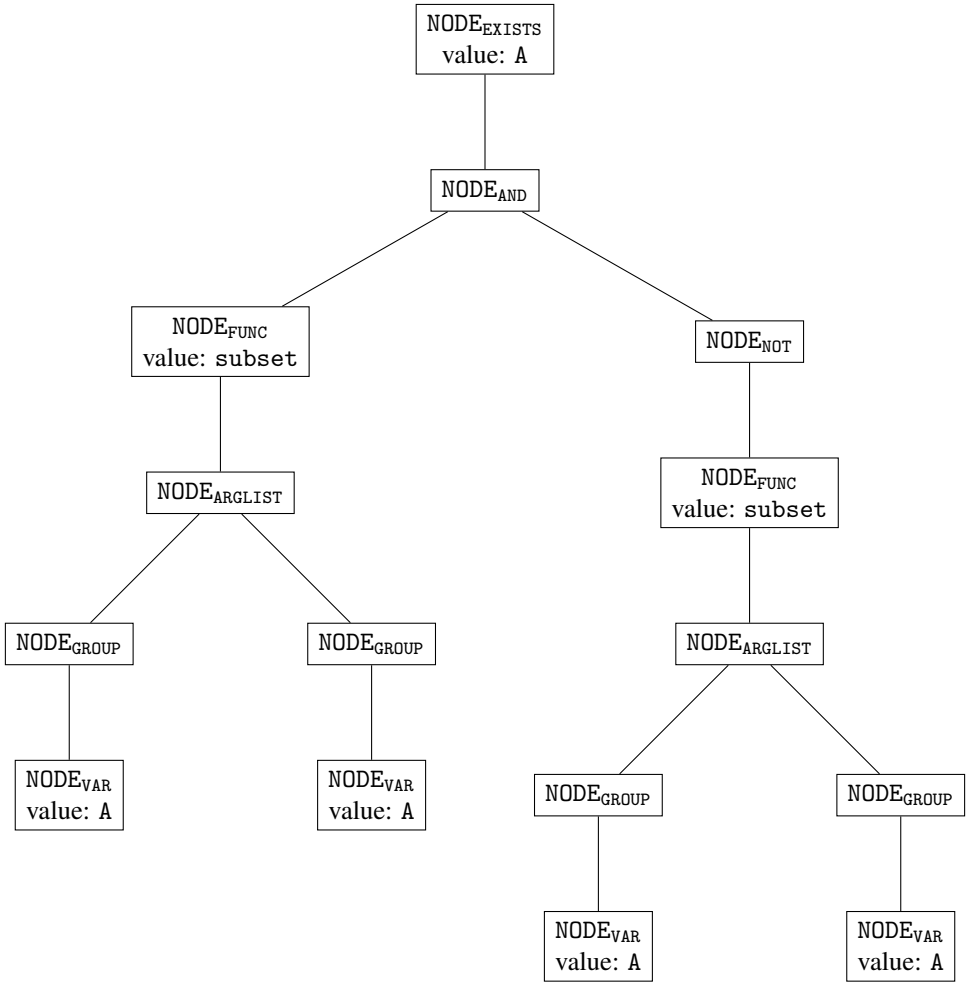

Figure 9: Simplified AST for the expression $\exists A\ \big(\mathrm{subset}(A,A) \wedge \neg\big(\mathrm{subset}(A,A)\big)\big)$.

3.6 Examples

3.6.1 Set Theory

To illustrate the complete decoding process, let us revisit the running example from Section 3.1, repeated here for convenience:

```
unsat
(declare-type $$unsorted ())
(declare-const tptp.subset (-> $$unsorted $$unsorted Bool))
(define @t1 () (eo::var "A" $$unsorted))
(define @t2 () (tptp.subset @t1 @t1))
(define @t3 () (@list @t1))
(define @t4 () (forall @t3 (=> @t2 @t2)))
(define @t5 () (forall @t3 true))
(assume @p1 (not @t4))
(assume @p2 (forall @t3 @t2))
; WARNING: add trust step for TRUST_THEORY_REWRITE
; trust TRUST_THEORY_REWRITE
(step @p3 :rule trust :premises () :args ((= (not true) false)))
; trust TRUST_THEORY_REWRITE
(step @p4 :rule trust :premises () :args ((= @t5 true)))
; trust TRUST_THEORY_REWRITE
(step @p5 :rule trust :premises () :args ((= @t4 @t5)))
(step @p6 :rule trans :premises (@p5 @p4))
(step @p7 :rule cong :premises (@p6) :args (not))
(step @p8 :rule trans :premises (@p7 @p3))
(step @p9 false :rule eq_resolve :premises (@p1 @p8))
```

After refactoring (see Section 3.2 and Listing 3), the proof is resolved line by line, resulting in Listing 4.

Listing 4: Resolved version of the proof for the set problem.

```
unsat
declare-type a
declare-const subset (-> a a Bool)
define A a
define (subset A A)
define (A)
define (forall (A) (=> (subset A A) (subset A A)))
define (forall (A) true)
assume (not (forall (A) (=> (subset A A) (subset A A))))
assume (forall (A) (subset A A))
step ((= (not true) false)) (trust)
step ((= (forall (A) true) true)) (trust)
step ((= (forall (A) (=> (subset A A) (subset A A)))
         (forall (A) true))) (trust)
step (trans (((= (forall (A) (=> (subset A A) (subset A A)))
                 (forall (A) true)))
             ((= (forall (A) true) true))))
step (not) (cong)
step (trans ((not) ((= (not true) false))))
step (eq_resolve ((not (forall (A) (=> (subset A A) (subset A A))))))
```

Listing 4 is significantly easier to understand than the original proof, yet it still requires some additional mental simplification to fully understand the reasoning. Finally, a fully simplified representation obtained after lexical analysis and parsing is presented in Listing 5.

Listing 5: Simplified version of the proof for the set problem.

```
unsat
declare-type a
declare-const subset (-> a a Bool)
define A a
define (subset A A)
define (A)
define (forall (A) (or (not (subset A A)) (subset A A)))
define (forall (A) true)
assume (exists (A) (and (subset A A) (not (subset A A))))
assume (forall (A) (subset A A))
step (= (not true) false) (trust)
step (= (forall (A) true) true) (trust)
step (= (forall (A) (or (not (subset A A)) (subset A A)))(forall (A) true)) (trust)
step (trans ((= (forall (A) (or (not (subset A A)) (subset A A))) (forall (A) true))
            (= (forall (A) true) true)))
step (not) (cong)
step (trans ((not) (= (not true) false)))
step (eq_resolve (exists (A) (and (subset A A) (not (subset A A)))))
```

Additionally, the decoder generates a formatted version of the proof, aligning the proof arguments and steps to the left and positioning the rules and premises on the right (see Listing 6). This layout aims to enhance readability and understanding of the reasoning process.

Listing 6: Formatted version of the proof for the set problem.

```
unsat
declare-type a
declare-const subset (-> a a Bool)
define A a
define (subset A A)
define (A)
define (forall (A) (or (not (subset A A))
                       (subset A A)))
define (forall (A) true)
assume (exists (A) (and (subset A A)
                        (not (subset A A))))
assume (forall (A) (subset A A))
step (= (not true) false)                    (trust)
step (= (forall (A) true) true)              (trust)
step (= (forall (A) (or (not (subset A A))
                        (subset A A)))
        (forall (A) true))                   (trust)
step                                         (trans ((= (forall (A) (or (not (subset A A)) (subset A A)))
                                                        (forall (A) true))
                                                    (= (forall (A) true) true)))
step (not)                                   (cong)
step                                         (trans ((not) (= (not true) false)))
step                                         (eq_resolve (exists (A) (and (subset A A)
                                                                          (not (subset A A)))))
```

3.6.2 Linear Integer Arithmetic

To illustrate `cvc5`'s proof representation in the theory of Linear Integer Arithmetic (LIA), consider the following problem:

$$\forall x \in \mathbb{Z}: \quad x \neq 0 \qquad \text{(Axiom 1)}$$
$$\exists x \in \mathbb{Z}: \quad x = 0 \qquad \text{(Axiom 2)}$$

This problem is unsatisfiable because the first axiom states that for every integer $x \in \mathbb{Z}$, x is not equal to 0, while the second axiom claims that there exists an integer x for which $x = 0$. The inherent contradiction between these two assertions makes the overall formula unsatisfiable. The SMT-LIB representation of this problem is shown in Listing 7.

Listing 7: SMT-LIB representation of the LIA problem.

```
(set-logic LIA)
(assert (forall ((x Int)) (not (= x 0))))
(assert (exists ((x Int)) (= x 0)))
(check-sat)
```

Using this input, the solver generates a proof that the problem is unsatisfiable (see Listing 8. Like the set theory example, the proof undergoes the stages refactoring (see Listing 9), resolving (see Listing 10), parsing (see Listing 11), and formatting (see Listing 12).

Listing 8: Proof production of the LIA problem.

```
unsat
(define @t1 () (eo::var "x" Int))
(define @t2 () (= @t1 0))
(define @t3 () (@list @t1))
(define @t4 () (forall @t3 (not @t2)))
(define @t5 () (= 0 0))
(assume @p1 @t4)
(assume @p2 (exists @t3 @t2))
; WARNING: add trust step for TRUST_THEORY_REWRITE
; trust TRUST_THEORY_REWRITE
(step @p3 :rule trust :premises () :args ((= (not true) false)))
; trust TRUST_THEORY_REWRITE
(step @p4 :rule trust :premises () :args ((= @t5 true)))
(step @p5 :rule cong :premises (@p4) :args (not))
(step @p6 :rule trans :premises (@p5 @p3))
; trust TRUST_THEORY_REWRITE
(step @p7 :rule trust :premises () :args ((= @t4 (not @t5))))
(step @p8 :rule trans :premises (@p7 @p6))
(step @p9 false :rule eq_resolve :premises (@p1 @p8))
```

Listing 9: Refactored version of the proof for the LIA problem.

```
unsat
(define @t1 () (eo::var "x" Int))
(define @t2 () (= @t1 0))
(define @t3 () (@t1))
(define @t4 () (forall @t3 (not @t2)))
```

```
(define @t5 () (= 0 0))
(assume @p1 @t4)
(assume @p2 (exists @t3 @t2))
(step @p3 :rule trust :premises () :args ((= (not true) false)))
(step @p4 :rule trust :premises () :args ((= @t5 true)))
(step @p5 :rule cong :premises (@p4) :args (not))
(step @p6 :rule trans :premises (@p5 @p3))
(step @p7 :rule trust :premises () :args ((= @t4 (not @t5))))
(step @p8 :rule trans :premises (@p7 @p6))
(step @p9 false :rule eq_resolve :premises (@p1 @p8))
```

Listing 10: Resolved version of the proof for the set problem.

```
unsat
define (x) Int
define (= (x) 0)
define (x)
define (forall (x) (not (= (x) 0)))
define (= 0 0)
assume (forall (x) (not (= (x) 0)))
assume (exists (x) (= (x) 0))
step (= (not true) false) (trust)
step (= (= 0 0) true) (trust)
step (not) (cong ((= (= 0 0) true)))
step (trans ((not) (= (not true) false)))
step (= (forall (x) (not (= (x) 0))) (not (= 0 0))) (trust)
step (trans ((= (forall (x) (not (= (x) 0))) (not (= 0 0))) ((not) (= (not true) false))))
step (eq_resolve ((forall (x) (not (= (x) 0))) ((= (forall (x) (not (= (x) 0))) (not (= 0 0)))
                  ((not) (= (not true) false)))))
```

Listing 11: Simplified version of the proof for the LIA problem.

```
unsat
define (x) Int
define (= (x) 0)
define (x)
define (forall (x) (not (= (x) 0)))
define (= 0 0)
assume (forall (x) (not (= (x) 0)))
assume (exists (x) (= (x) 0))
step (= (not true) false) (trust)
step (= (= 0 0) true) (trust)
step (not) (cong ((= (= 0 0) true)))
step (trans ((not) (= (not true) false)))
step (= (forall (x) (not (= (x) 0))) (not (= 0 0))) (trust)
step (trans ((= (forall (x) (not (= (x) 0))) (not (= 0 0))) ((not) (= (not true) false))))
step (eq_resolve ((forall (x) (not (= (x) 0))) ((= (forall (x) (not (= (x) 0))) (not (= 0 0)))
                  ((not) (= (not true) false)))))
```

Listing 12: Formatted version of the proof for the LIA problem.

```
unsat
unsat
define (x) Int
```

```
define (= (x) 0)
define (x)
define (forall (x) (not (= (x) 0)))
define (= 0 0)
assume (forall (x) (not (= (x) 0)))
assume (exists (x) (= (x) 0))
step (= (not true) false)                                  (trust)
step (= (= 0 0) true)                                      (trust)
step (not)                                                 (cong ((= (= 0 0) true)))
step                                                       (trans ((not) (= (not true) false)))
step (= (forall (x) (not (= (x) 0))) (not (= 0 0)))        (trust)
step                                                       (trans ((= (forall (x) (not (= (x) 0))) (not (= 0 0)))
                                                                  ((not) (= (not true) false))))
step                                                       (eq_resolve ((forall (x) (not (= (x) 0)))
                                                                       ((= (forall (x) (not (= (x) 0)))
                                                                           (not (= 0 0)))
                                                                       ((not) (= (not true) false)))))
```

4 Implementation

4.1 Installation

The project was developed in the C programming language, following the C17 standard, and compiled using GCC version 11.4.0. All source code is available at `https://github.com/ooezenoglu/cvc5-interpreter`. The program is build by running the command `make` from within the terminal, which links and compiles all the required files. To run the program, the user must call the executable `./cvc5-interpreter` along with one or a sequence of the following flags listed in Table 3, depending on the intended use-case. For example, a TPTP problem can be translated into an SMT-LIB problem (by the `cvc5` parser), executed by the `cvc5` solver and finally the proof decoded with the following command:

```
make && ./cvc5-interpreter
--p --f <PATH-TPTP-PROBLEM> --c <PATH-cvc5-PARSER> --r <PATH-cvc5-SOLVER> --d
```

Flag	Description
`--p` (optional)	Parse the TPTP problem given in `--f` into SMT-LIB format
`--c`	Required when `--p` is set; Relative path to the `cvc5` parser
`--d` (optional)	Calls the `cvc5` solver and decodes the proof
`--r`	Required when `--d` is set; Relative path to the `cvc5` solver

Table 3: Command-line flags

4.2 Decoder-Modules

As detailed in Section 3, a `cvc5` proof is decoded in four main steps. To reflect these steps, the decoder is organized in a modular fashion. The four core modules are depicted in Figure 10. The following sections provide an explanation of each module.

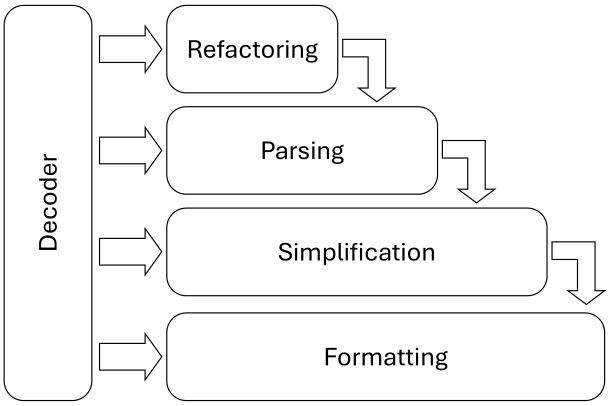

Figure 10: Modules of the decoder.

4.2.1 Refactoring

The refactoring module (see also Section 3.2) converts a raw `cvc5` proof file into a clean version for further processing. It reads the file line by line, removes trailing newlines and comment lines, and cleans each line by stripping TPTP residues, extra characters, and spaces. For type declaration lines, it extracts the original type name, generates a new type variable, and updates the mapping so that all occurrences are replaced. The new proof is stored in a file with suffix "_refactored.txt".

4.2.2 Parsing and Simplification

The core of the decoder is the parser module, which takes a refactored proof and outputs a parsed version of the proof (saved with the suffix "_parsed.txt"). In a similar fashion, the simplification module produces a fully simplified proof (saved with the suffix "_simplified.txt"). The parser extracts the proof components from each refactored proof line as described in Section 3. These components are stored in a hashmap, where each proof line's tag serves as the key and the corresponding data structure containing its components is the value.[10]

Suppose a proof line reads `(assume @p1 (not @t4))`. The parser stores `@p1` as the key and `assume` as the type. The argument `(not @t4)` references a tag of another line. Therefore, the parser module looks up `@t4` in the hashmap and replaces `(not @t4)` with the argument associated with `@t4`. Because the line keyed by `@t4` was already extracted and simplified, only including `@t4` completes the argument resolution for `@p1`. This avoids recursively adding duplicate entries. Once the logical expression is resolved, it is passed to the simplification module, which runs the lexical analysis and the parser for simplification (see below). After `@p1` is resolved and simplified, the resulting entry in the hashmap is stored as follows:[11]

```
Key (Tag):      @p1
Value (Line):
                Type:       assume
                Args:       (not @t4)
```

[10] A hashmap was chosen because it provides efficient insertion and retrieval operations.

[11] Note that other components (like rule and premises) are also stored in the hashmap but are omitted here, as they are empty in this example.

```
Resolved:    (not (forall (A) (=> (subset (A) (A)) (subset (A) (A)))))
Simplified:  (exists (A) (and (subset (A) (A)) (not (subset (A) (A)))))
```

To implement the lexer (for lexical analysis) as described in Section 3.4 and the parser as described in Section 3.5, this work uses the popular compiler construction tools Flex and Bison, respectively (Thain 2020). The interaction of these tools for processing a logical expression is described in Algorithm 1.

Algorithm 1: Simplification Process via Lexer and Parser

Data: Resolved logical expression as string
Result: String representation of the simplified logical expression

Lexer (Flex) Scan the input string and dissect it into tokens.
Parser (Bison): Construct the AST from the tokens.
Parser (Bison): Recursively traverse the AST and apply simplifications.
repeat
  Convert implications;
  Eliminate double negations;
  Transform negated quantifiers;
  Apply De Morgan's laws;
until *no transformation applies*;
Output:
Return the final simplified AST as string.

4.2.3 Formatting

The formatting module formats each simplified proof line by concatenating the type, arguments, and possibly a note with the line's rule and the premises. It adds padding so that the rule and premises are right-aligned, similar to traditional mathematical proofs. This right alignment makes the proof more visually organized and easier to review. Each formatted line is printed and then written to an output file ending on "_formatted.txt".

5 Evaluation

The decoder currently supports only a subset of the theories offered by `cvc5`. In particular, it handles linear integer arithmetic, linear real arithmetic, non-linear integer arithmetic, non-linear real arithmetic, and sets. However, because each theory produces varying proof patterns, the decoder cannot yet capture all the theories in `cvc5`. Due to its modular design extending support to additional theories (e.g., bit vectors, arrays) or even higher-order logic requires only adjustments to the lexer and parser rules.

To evaluate the decoder's effectiveness, both self-generated tests and publicly available benchmarks were used. The benchmarks consisted of a collection of first-order logic problems from SMT-LIB Release 2023 (Preiner et al. 2024). In total, 36 proof files were examined, covering SMT-LIB theories such as LIA, LRA, NIA, and NRA. Unfortunately, benchmarks yielding unsatisfiable

results were not available for theories involving floating-point arithmetic (e.g., FP or FPLRA), therefore rendering them unsuitable for evaluation. Additionally, because the decoder's main function is to simplify logical expressions, 30 test expressions were designed to verify that certain expressions, including negated quantifiers, nested expressions, numbers, and function applications, are properly simplified. Furthermore, 10 invalid logical expressions were tested to ensure that the grammar correctly rejects faulty input expressions (e.g., missing brackets).

Upon examining the decoded proofs, it becomes evident that they are significantly more readable and comprehensible than the original proofs, primarily due to the applied simplifications and formatting. However, this improvement diminishes with larger proofs that contain an excessive number of tags. In such cases, the decoder often fails to give any results. Moreover, when proofs do include many tags in the arguments correctly resolved by the decoder, it sometimes still ceases to simplify the expressions, presumably because the AST structure becomes excessively deep or nested. Nonetheless, all test expressions have been processed correctly by the decoder (that is, either simplified or appropriately rejected) indicating that the simplification rules are correctly implemented. Overall, the decoder performs very well for smaller to medium-sized proofs.

The results demonstrate the decoder's current capabilities and confirm its reliability in handling a variety of small proof scenarios. Due to variations across different theories and also within a single theory, the decoder's grammar must be continuously updated. Future work will involve extending support to additional theories (such as Bit-vectors and arrays) and conducting broader evaluations, both in terms of complexity and proof size, to ensure general applicability and robustness.

6 Future Work

The current work lays the foundation for several promising future research directions. Currently, the decoder only supports a limited set of theories. Since `cvc5` supports a broad range of theories, however, it would be worthwhile to examine their proof structures and extend the grammar accordingly. That way, proofs in additional theories can become more user-friendly. Moreover, even within the same theory, proof structures can vary. For example, some proofs use quantification over multiple variables, which the decoder currently cannot handle. Adding support for new theories is straightforward, since only the lexer and parser rules need to be updated.

Furthermore, the decoder currently performs simplifications on expressions limited to logical transformations, for instance, applying De Morgan's laws. A natural next step would be to extend these simplifications to the theory level. In this approach, the decoder could first execute the standard logical simplifications and then further simplify the expression based on the specific theory. For example, when working within the theory of linear integer arithmetic, the decoder might convert expressions like `(not (>= (x) 4))` directly into the more intuitive form `(< (x) 4)`. This additional layer of theory-specific simplifications would not only enhance the clarity of the proofs but also enhance alignment with the conventional representations in mathematical literature.

Another possible step would be to offer a problem encoder in addition to the proof decoder, i.e., one that brings the input problems into a more readable form. Since the problems provided to `cvc5` are formulated in the special TPTP / SMT-LIB syntax, they, too, are often difficult to understand. Thus, a similar approach could be applied to the input problems by identifying key structures and defining grammars for them. This would result in a comprehensive tool capable of intuitively representing both inputs and outputs, thereby significantly streamlining the whole

problem-solving workflow.

Additionally, this thesis project has so far concentrated exclusively on proofs formulated in first-order logic. However, the `cvc5` solver is one of the few systems capable of generating proofs for problems expressed in higher-order logic. A natural extension of this work would therefore be to broaden the framework to support proofs in higher-order logic as well. Such an expansion would not only extend the applicability of the system to a wider range of logical problems, but also take full advantage of `cvc5`'s capabilities. Exploring the structural differences and additional complexity in higher-order logical proofs could provide valuable insights and further improve the robustness and flexibility of the decoder.

Furthermore, a promising next step would be to analyze `cvc5`'s internal proof production logic in detail and to identify elements that, while crucial for internal processing, are not significant for human readers. For example, the expression `(= (not true) false)` is generally understood and could thus be omitted from the proof. Removing these redundant or trivial statements would lead to more concise proofs, improving readability, reducing complexity, and enabling users to follow the core logical arguments more easily.

Currently, proofs are generated and displayed in prefix notation. However, representing them in infix notation may be more natural, as it aligns more closely with the conventions in mathematical proofs. Since the abstract syntax tree for each argument is already set up, adapting the tree traversal to output proofs in an infix format should be relatively straightforward. Such a modification would primarily involve changes to the formatting logic rather than structural adjustments to the parser module of the decoder.

An interesting next step would be to show the decoded proof directly in LaTeX for a clearer presentation. To do this, the abstract syntax tree could be recursively traversed where each node type is mapped to the relevant LaTeX macro. That way, logical operators, quantifiers, and other elements (e.g., $\forall, \neg, \wedge$) would appear in a mathematical notation. In the end, LaTeX code could be produced which can be embedded in a document and compiled, showing a reader-friendly proof.

Last but not least, a graphical tree representation of the proof could make its overall structure much clearer. Internally, the proof is already organized as an abstract syntax tree during simplification, which is converted back to its string representation. A next step could be to visualize this tree in a user-friendly graphical view, with each node showing a different part of the proof. This would likely simplify debugging and deepen the understanding of more complex proofs. Adding interactive features like node expansion could further improve navigation through detailed proofs.

7 Conclusion

In conclusion, this thesis shows a step forward in turning convoluted, machine-generated proofs from the SMT solver `cvc5` into clear, human-friendly formats. A decoder was built that processes proofs in four stages: First, the raw proof is cleaned up for further processing (refactoring). Next, the main components are extracted and cross-references are resolved (parsing). Then, lexical analysis is combined with AST construction and simplification while preserving the overall logical structure (simplifying). And finally, the output is formatted (formatting).

The tests have shown that the decoder works very well on smaller to medium-sized proofs. However, with larger proofs that have many layers of nesting, there are some challenges. These issues suggest that extending the decoder's grammar and extensive testing on more problems

could be helpful. It might also be useful to explore other output formats, such as using standard mathematical notation, LaTeX extensions, or even graphical representations, to further establish more traditional logical proof representations.

Overall, this work demonstrates that techniques from compiler construction, such as tokenization and creating abstract syntax trees, can greatly improve the way automated proofs are presented. The modular design of the decoder makes it easy to add support for more theories and even higher-order logic in the future. This thesis lays the foundation towards a deeper understanding of the reasoning process, to easier verification, and greater trust in automated reasoning tools.

List of Figures

List of Tables

Listings

References

BACKES, JOHN; PAULINE BOLIGNANO; BYRON COOK; CATHERINE DODGE; ANDREW GACEK; KASPER LUCKOW; NEHA RUNGTA; OKSANA TKACHUK; und CARSTEN VARMING. 2018. Semantic-based automated reasoning for aws access policies using smt. *2018 formal methods in computer aided design (fmcad)*, 1–9.

BARBOSA, HANIEL; CLARK W. BARRETT; MARTIN BRAIN; GEREON KREMER; HANNA LACHNITT; MAKAI MANN; ABDALRHMAN MOHAMED; MUDATHIR MOHAMED; AINA NIEMETZ; ANDRES NÖTZLI; ALEX OZDEMIR; MATHIAS PREINER; ANDREW REYNOLDS; YING SHENG; CESARE TINELLI; und YONI ZOHAR. 2022. cvc5: A versatile and industrial-strength SMT solver. *Tools and algorithms for the construction and analysis of systems - 28th international conference, TACAS 2022, held as part of the european joint conferences on theory and practice of software, ETAPS 2022, munich, germany, april 2-7, 2022, proceedings, part I*, hrsg. von Dana Fisman und Grigore Rosu, *Lecture Notes in Computer Science*, Ausg. 13243, 415–442. Springer. URL `https://doi.org/10.1007/978-3-030-99524-9_24`.

BARBOSA, HANIEL; ANDREW REYNOLDS; DANIEL EL OURAOUI; CESARE TINELLI; und CLARK BARRETT. 2019. Extending SMT solvers to higher-order logic. *Proceedings of the 27th international conference on automated deduction (cade '19)*, hrsg. von Pascal Fontaine, *Lecture Notes in Artificial Intelligence*, Ausg. 11716, 35–54. Springer. Natal, Brazil. URL `http://theory.stanford.edu/~barrett/pubs/BREO+19.pdf`.

BARRETT, CLARK; CHRISTOPHER L. CONWAY; MORGAN DETERS; LIANA HADAREAN; DEJAN JOVANOVIĆ; TIM KING; ANDREW REYNOLDS; und CESARE TINELLI. 2011. Cvc4. *Computer aided verification*, hrsg. von Ganesh Gopalakrishnan und Shaz Qadeer, 171–177. Berlin, Heidelberg: Springer Berlin Heidelberg.

BARRETT, CLARK; PASCAL FONTAINE; und CESARE TINELLI. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`.

BARRETT, CLARK; DANIEL KROENING; und TOM MELHAM. 2014. Problem solving for the 21st century: Efficient solvers for satisfiability modulo theories. Tech. Rep. 3, London Mathematical Society and the Smith Institute for Industrial Mathematics and System Engineering.

BARRETT, CLARK; LEONARDO DE MOURA; und PASCAL FONTAINE. 2015. Proofs in satisfiability modulo theories. *All about proofs, proofs for all*, hrsg. von David Delahaye und Bruno Woltzenlogel Paleo, *Mathematical Logic and Foundations*, Ausg. 55, 23–44. London, UK: College Publications. URL `http://theory.stanford.edu/~barrett/pubs/BdMF15.pdf`.

BARRETT, CLARK; ROBERTO SEBASTIANI; SANJIT A. SESHIA; und CESARE TINELLI. 2009. *Satisfiability modulo theories*, 825–885. Nr. 1 in Frontiers in Artificial Intelligence and Applications, 1 Ed. IOS Press.

TER BEEK, MAURICE H.; ROD CHAPMAN; RANCE CLEAVELAND; HUBERT GARAVEL; RONG GU; IVO TER HORST; JEROEN J. A. KEIREN; THIERRY LECOMTE; MICHAEL LEUSCHEL; KRISTIN YVONNE ROZIER; AUGUSTO SAMPAIO; CRISTINA SECELEANU; MARTYN THOMAS;

TIM A. C. WILLEMSE; und LIJUN ZHANG. 2024. Formal methods in industry. *Form. Asp. Comput.* 37. URL https://doi.org/10.1145/3689374.

BONACINA, MARIA PAOLA. 2018. On conflict-driven reasoning.

BOUTON, THOMAS; DIEGO CAMINHA B. DE OLIVEIRA; DAVID DÉHARBE; und PASCAL FONTAINE. 2009. verit: An open, trustable and efficient smt-solver. *Automated deduction – cade-22*, hrsg. von Renate A. Schmidt, 151–156. Berlin, Heidelberg: Springer Berlin Heidelberg.

BUSS, SAMUEL. 2012. *bussproofs – a latex package for typesetting proof trees*. Available at https://ctan.org/pkg/bussproofs.

CHIMISLIU, VALENTIN, und FRANZ WOTAWA. 2012. Category partition method and satisfiability modulo theories for test case generation. *Proceedings of the 7th international workshop on automation of software test*, AST '12, 64–70. IEEE Press.

CHRIST, JÜRGEN; JOCHEN HOENICKE; und ALEXANDER NUTZ. 2012. Smtinterpol: An interpolating smt solver. *Model checking software*, hrsg. von Alastair Donaldson und David Parker, 248–254. Berlin, Heidelberg: Springer Berlin Heidelberg.

CIMATTI, ALESSANDRO; ALBERTO GRIGGIO; BASTIAAN JOOST SCHAAFSMA; und ROBERTO SEBASTIANI. 2013. The mathsat5 smt solver. *Tools and algorithms for the construction and analysis of systems*, hrsg. von Nir Piterman und Scott A. Smolka, 93–107. Berlin, Heidelberg: Springer Berlin Heidelberg.

COOK, STEPHEN A. 1971. The complexity of theorem-proving procedures. *Proceedings of the third annual acm symposium on theory of computing*, STOC '71, 151–158. New York, NY, USA: Association for Computing Machinery. URL https://doi.org/10.1145/800157.805047.

DAVIS, MARTIN; GEORGE LOGEMANN; und DONALD LOVELAND. 1962. A machine program for theorem-proving. *Commun. ACM* 5.394–397. URL https://doi.org/10.1145/368273.368557.

DAVIS, MARTIN, und HILARY PUTNAM. 1960. A computing procedure for quantification theory. *J. ACM* 7.201–215. URL https://doi.org/10.1145/321033.321034.

DE MOURA, LEONARDO, und NIKOLAJ BJØRNER. 2011. Satisfiability modulo theories: introduction and applications. *Commun. ACM* 54.69–77. URL https://doi.org/10.1145/1995376.1995394.

DETLEFS, DAVID; GREG NELSON; und JAMES B. SAXE. 2005. Simplify: a theorem prover for program checking. *J. ACM* 52.365–473. URL https://doi.org/10.1145/1066100.1066102.

DUTERTRE, BRUNO. 2014. Yices 2.2. *Computer aided verification*, hrsg. von Armin Biere und Roderick Bloem, 737–744. Cham: Springer International Publishing.

EÉN, NIKLAS, und NIKLAS SÖRENSSON. 2004. An extensible sat-solver. *Theory and applications of satisfiability testing*, hrsg. von Enrico Giunchiglia und Armando Tacchella, 502–518. Berlin, Heidelberg: Springer Berlin Heidelberg.

FAN, HONGYU; WEITING LIU; und FEI HE. 2022. Interference relation-guided smt solving for multi-threaded program verification. *Proceedings of the 27th acm sigplan symposium on principles and practice of parallel programming*, PPoPP '22, 163–176. New York, NY, USA: Association for Computing Machinery. URL https://doi.org/10.1145/3503221.3508424.

FARIAS, BRUNO; RAFAEL MENEZES; EDDIE B. DE LIMA FILHO; YOUCHENG SUN; und LUCAS C. CORDEIRO. 2024. Esbmc-python: A bounded model checker for python programs. *Proceedings of the 33rd acm sigsoft international symposium on software testing and analysis*, ISSTA 2024, 1836–1840. New York, NY, USA: Association for Computing Machinery. URL https://doi.org/10.1145/3650212.3685304.

FELBINGER, HERMANN, und CHRISTIAN SCHWARZL. 2014. Suitability analysis of csp- and smt-solvers for test case generation. *Proceedings of the 6th international workshop on constraints in software testing, verification, and analysis*, CSTVA 2014, 40–49. New York, NY, USA: Association for Computing Machinery. URL https://doi.org/10.1145/2593735.2593741.

GODEFROID, PATRICE; MICHAEL Y. LEVIN; und DAVID MOLNAR. 2012. Sage: whitebox fuzzing for security testing. *Commun. ACM* 55.40–44. URL https://doi.org/10.1145/2093548.2093564.

ITZHAKY, SHACHAR; HILA PELEG; NADIA POLIKARPOVA; REUBEN N. S. ROWE; und ILYA SERGEY. 2021. Cyclic program synthesis. *Proceedings of the 42nd acm sigplan international conference on programming language design and implementation*, PLDI 2021, 944–959. New York, NY, USA: Association for Computing Machinery. URL https://doi.org/10.1145/3453483.3454087.

KUGELE, STEFAN; GHEORGHE PUCEA; RAMONA POPA; LAURENT DIEUDONNÉ; und HORST ECKARDT. 2015. On the deployment problem of embedded systems. *Proceedings of the 2015 acm/ieee international conference on formal methods and models for codesign*, MEMOCODE '15, 158–167. USA: IEEE Computer Society. URL https://doi.org/10.1109/MEMCOD.2015.7340482.

KULIK, TOMAS; BRIJESH DONGOL; PETER GORM LARSEN; HUGO DANIEL MACEDO; STEVE SCHNEIDER; PETER W. V. TRAN-JØRGENSEN; und JAMES WOODCOCK. 2022. A survey of practical formal methods for security. *Form. Asp. Comput.* 34. URL https://doi.org/10.1145/3522582.

LEE, JIA; GEUNYEOL YU; und KYUNGMIN BAE. 2022. Efficient smt-based model checking for signal temporal logic. *Proceedings of the 36th ieee/acm international conference on automated software engineering*, ASE '21, 343–354. IEEE Press. URL https://doi.org/10.1109/ASE51524.2021.9678719.

LEINO, K. RUSTAN M. 2010. Dafny: An automatic program verifier for functional correctness. *Logic for programming, artificial intelligence, and reasoning*, hrsg. von Edmund M. Clarke und Andrei Voronkov, 348–370. Berlin, Heidelberg: Springer Berlin Heidelberg.

LEVIN, L. A. 1973. Universal sequential search problems. *Probl. Peredachi Inf.* 9.115–116, Translation: *Problems Inform. Transmission*, 1973, vol. 9, no. 3, pp. 265–266.

MathNet: http://mi.mathnet.ru/ppi914, MathSciNet: http://mathscinet.ams.org/mathscinet-getitem?mr=340042, zbMATH: https://zbmath.org/?q=an:0313.02026.

DE MOURA, LEONARDO, und NIKOLAJ BJØRNER. 2008. Z3: An efficient smt solver. *Tools and algorithms for the construction and analysis of systems*, hrsg. von C. R. Ramakrishnan und Jakob Rehof, 337–340. Berlin, Heidelberg: Springer Berlin Heidelberg.

NIEMETZ, AINA, und MATHIAS PREINER. 2020. Bitwuzla at the smt-comp 2020. URL https://arxiv.org/abs/2006.01621.

NIEMETZ, AINA; MATHIAS PREINER; CLIFFORD WOLF; und ARMIN BIERE. 2018. Btor2 , btormc and boolector 3.0. *Computer aided verification*, hrsg. von Hana Chockler und Georg Weissenbacher, 587–595. Cham: Springer International Publishing.

NIEUWENHUIS, ROBERT; ALBERT OLIVERAS; und CESARE TINELLI. 2006. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM* 53.937–977. URL https://doi.org/10.1145/1217856.1217859.

PEREIRA, PHILLIPE; HIGO ALBUQUERQUE; HENDRIO MARQUES; ISABELA SILVA; CELSO CARVALHO; LUCAS CORDEIRO; VANESSA SANTOS; und RICARDO FERREIRA. 2016. Verifying cuda programs using smt-based context-bounded model checking. *Proceedings of the 31st annual acm symposium on applied computing*, SAC '16, 1648–1653. New York, NY, USA: Association for Computing Machinery. URL https://doi.org/10.1145/2851613.2851830.

PREINER, MATHIAS; HANS-JÖRG SCHURR; CLARK BARRETT; PASCAL FONTAINE; AINA NIEMETZ; und CESARE TINELLI. 2024. Smt-lib release 2023 (non-incremental benchmarks). URL https://doi.org/10.5281/zenodo.10607722.

REPS, THOMAS. 1998. "maximal-munch" tokenization in linear time. *ACM Trans. Program. Lang. Syst.* 20.259–273. URL https://doi.org/10.1145/276393.276394.

STUMP, AARON; CLARK W. BARRETT; und DAVID L. DILL. 2002. Cvc: A cooperating validity checker. *Computer aided verification*, hrsg. von Ed Brinksma und Kim Guldstrand Larsen, 500–504. Berlin, Heidelberg: Springer Berlin Heidelberg.

THAIN, DOUGLAS. 2020. *Introduction to compilers and language design*. 2nd Ed. Revision Date: January 15, 2021. URL http://compilerbook.org.

TODOROV, VASSIL; FRÉDÉRIC BOULANGER; und SAFOUAN TAHA. 2018. Formal verification of automotive embedded software. *Proceedings of the 6th conference on formal methods in software engineering*, FormaliSE '18, 84–87. New York, NY, USA: Association for Computing Machinery. URL https://doi.org/10.1145/3193992.3194003.

WANG, HERAN; XIAOGANG DONG; BIN GU; XIAOFENG LI; und RUIMING ZHONG. 2024. Improve software development: An overview of program synthesis. *Proceedings of the 2023 5th international conference on software engineering and development*, ICSED '23, 1–7. New York, NY, USA: Association for Computing Machinery. URL https://doi.org/10.1145/3637792.3637793.

WASSYNG, ALAN; MARK S. LAWFORD; und THOMAS S.E. MAIBAUM. 2011. Software certification experience in the canadian nuclear industry: lessons for the future. *Proceedings of the ninth acm international conference on embedded software*, EMSOFT '11, 219–226. New York, NY, USA: Association for Computing Machinery. URL https://doi.org/10.1145/2038642.2038676.

XUE, ZHIYI; LIANGGUO LI; SENYUE TIAN; XIAOHONG CHEN; PINGPING LI; LIANGYU CHEN; TINGTING JIANG; und MIN ZHANG. 2024. Llm4fin: Fully automating llm-powered test case generation for fintech software acceptance testing. ISSTA 2024, 1643–1655. New York, NY, USA: Association for Computing Machinery. URL https://doi.org/10.1145/3650212.3680388.