Solving the Frobenius Problem in Z3: Exploring Quantifier Elimination

Paul Anton[1]
Supervisor(s): Soham Chakraborty[1]**, Dennis Sprokholt**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025

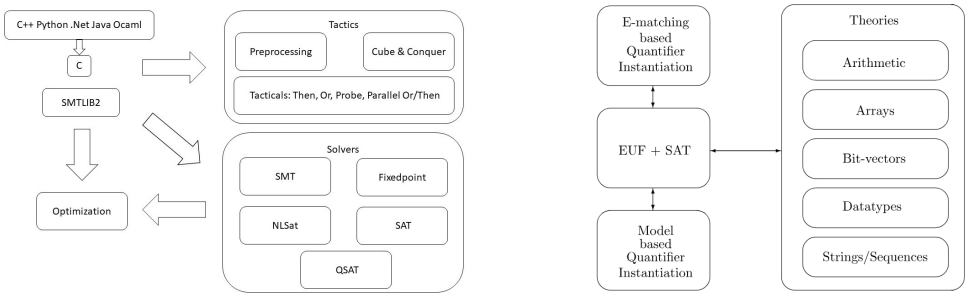
Name of the student: Paul Anton
Final project course: CSE3000 Research Project
Thesis committee: Soham Chakraborty, Dennis Sprokholt, Andy Zaidman

Abstract

Quantified formulas over linear integer arithmetic (LIA) are common in formal verification, yet they present significant challenges for satisfiability modulo theories (SMT) solvers such as Z3. In this paper, we explore whether quantifier elimination can improve solver performance on the 2-coin Frobenius Coin Problem, a benchmark involving quantified LIA formulas with structural simplicity. While Z3's default strategy relies on solving satisfiability of a quantified formula directly, we evaluate an alternative tactic-based approach using the `qe_rec` tactic to eliminate quantifiers first and produce a quantifier-free formula, which is then solved by the general purpose SMT solver. We conduct benchmarking across 54 satisfiable instances involving pairs of prime coin denominations. Our results indicate that quantifier elimination achieves better performance in both runtime and memory usage, solving more instances and offering consistent speed-ups of up to 10× compared to the default strategy.

1 Introduction

Satisfiability Modulo Theories (SMT) solving is more important than ever for formal verification and automated reasoning. For example, Amazon reported to generating over a billion SMT queries per day in 2022 [1]. SMT generalizes the Boolean satisfiability problem (SAT) [2] by having the ability to reason over more complex theories such as linear integer arithmetic, arrays, bit-vectors, and others. Z3 [3] is an efficient SMT solver developed by Microsoft Research and has become an integral part in numerous tools and frameworks used in both academia and industry. In software verification, Z3 is used in tools like Boogie [4], Viper [5], and Dafny [6], which prove the correctness of programs against formal specifications. Below there are 2 figures, the one on the left depicting the main components of Z3 and the one on the right depicting the architecture of the SMT solver.



(a) Overall system architecture of Z3 (b) Architecture of Z3's SMT Core solver

Figure 1: Z3 solver architecture diagrams [7]

Figure 1a presents an overview of the Z3 architecture. Z3 offers language bindings for several front-end interfaces, including `C++`, `Python`, `.NET`, `Java`, and `OCaml`. These APIs communicate with Z3's internal formula representation. Moreover, Z3 also accepts input in the standardized `SMTLIB2` format for compatibility and benchmarking purposes [8]. The input is processed through a combination of **tactics** and **solvers**. Tactics in Z3 are strategies that transform or reduce logical goals, typically sets of formulas, into simpler subgoals, optionally solving them. They form the basis for flexible proof and solving pipelines. For example, the tactic `simplify` applies algebraic simplifications to logical formulas, while `smt`

applies a SAT-based SMT solver to attempt to solve the formula directly. These tactics can be chained or combined using **tacticals**, which are higher-order operators that combine tactics in various ways. Notable tacticals include `then` (which composes tactics sequentially), `or_else` (which tries the first tactic and falls back to the second on failure), and `par-or` (which applies tactics in parallel and returns the result of the first to succeed). Once preprocessed, the problem is handed off to specialized solvers, including `SMT`, `SAT`, `NLSat`, `Fixedpoint` and `QSAT` for quantifiers. Z3 also provides support for optimization objectives, allowing cost-aware reasoning in satisfiability queries.

Figure 1b depicts the architecture of Z3's SMT core. At its center is a combination of a propositional `SAT` solver with `EUF` (Equality over Uninterpreted Functions), which forms the basis of logical reasoning in Z3. This core communicates with various **theory solvers** including arithmetic, arrays, bit-vectors, datatypes, and strings/sequences allowing Z3 to support a wide range of theories. For handling quantifiers ($\exists$ and $\forall$), Z3's SMT solver employs both E-matching-based and model-based quantifier instantiation techniques. Although both techniques are part of the SMT Core solver, Z3 primarily relies on the QSAT solver for handling quantified formulas.

When given a satisfiability query, Z3 returns one of three possible outcomes: `sat`, `unsat`, or `unknown`. The result `sat` indicates that there exists a model, which is an assignment of values to variables that satisfy all the constraints in the formula. In contrast, `unsat` means that no such assignment exists. The result `unknown` signifies that the solver was unable to determine the satisfiability status, which typically occurs due to timeouts being set.

Below are two small SMT-LIB examples. The first example is `sat`, as the integer assignment $x = 1$, $y = 1$ satisfies the equation $2x + 3y = 5$. The second is `unsat`, as no positive integers x and y satisfy $2x + 3y = 1$.

Listing 1: A satisfiable query

```
(set-logic QF_LIA)

(declare-const x Int)
(declare-const y Int)

(assert (> x 0))
(assert (> y 0))
(assert (= (+ (* 2 x) (* 3 y)) 5))

(check-sat)
```

Listing 2: An unsatisfiable query

```
(set-logic QF_LIA)

(declare-const x Int)
(declare-const y Int)

(assert (> x 0))
(assert (> y 0))
(assert (= (+ (* 2 x) (* 3 y)) 1))

(check-sat)
```

In both examples, the logic specified is `QF_LIA`, which stands for *Quantifier-Free Linear Integer Arithmetic*. Linear Integer Arithmetic (LIA), also known as Presburger Arithmetic (PA) [9], is a restricted fragment of arithmetic. It reasons over integers, but disallows multiplication between variables, with multiplication by constants being allowed. In other words, expressions like $2x + 3y = 5$ are valid, but expressions like $x \cdot y = 5$ or $x^2 = 4$ are not.

Quantified linear integer arithmetic (LIA) is relevant for many verification and reasoning tasks. A notable example is SQLSolver [10], a solver designed to verify the equivalence of SQL queries, a central challenge in database research. Reasoning over SQL requires additional expressiveness, which SQLSolver captures through LIA*, an extension of LIA. The solver translates LIA* formulas into standard LIA, which are then handled by Z3 to prove query equivalence. As noted by Reynolds et al., "SMT-based applications increasingly

rely on SMT solvers being able to deal with quantified formulas" [11], emphasizing the importance of advancing solver capabilities in this domain.

While more expressive, quantified formulas introduce significantly more complexity for SMT solvers, as the search space becomes effectively unbounded and solver heuristics must balance correctness, efficiency, and termination. For example, Z3's QSAT algorithm is only *partially correct*, that is, it may fail to terminate or return `unknown` even when the formula is satisfiable or unsatisfiable, but when it terminates, it does so with the correct result [12].

1.1 Frobenius Coin Problem

This added difficulty is reflected in solver performance. In [13], an alternative approach to solving LIA problems using finite automata is proposed, and the Frobenius coin problem [14] is used as a benchmark to compare the proposed method with solvers like Z3. In their evaluation, the automata-based approach outperforms Z3's implementation: out of 55 satisfiable instances of the Frobenius coin problem using 2 coins, Z3 times out on 51 of them, while their implementation times out on only 5. Princess [15], an SMT solver for Presburger Arithmetic, also outperformed Z3, timing out on just 13 instances.

The Frobenius Coin Problem is a classical problem in number theory. It asks: given coin denominations $c_1, c_2, \ldots, c_n$ and an infinite supply of coins with these denominations, what is the largest monetary sum that cannot be formed by any nonnegative linear combination of these coins? The problem can be encoded in `SMTLIB2` format for the 2 coin case by defining an integer P for which the following holds:

$$
\begin{aligned}
& P \geq 0 \quad \text{(P is non-negative)} \\
& \wedge \ \forall x_0, x_1.\ (x_0 \geq 0 \wedge x_1 \geq 0) \Rightarrow P \neq c_1 x_0 + c_2 x_1 \quad \text{(P is not representable)} \\
& \wedge \ \forall R.\ (\forall x_0, x_1.\ (x_0 \geq 0 \wedge x_1 \geq 0) \Rightarrow R \neq c_1 x_0 + c_2 x_1) \Rightarrow R \leq P \quad \text{(P is maximal)}
\end{aligned} \tag{1}
$$

Here, an unsatisfiable instance indicates that there is no largest non-representable amount, implying that the Frobenius number is infinite. Conversely, a satisfiable instance yields a value of P, which is the Frobenius number for the given coin pair. Throughout the remainder of this paper, we use the notation $\vec{c} = (c_1, c_2)$ to refer to such instances, where c_1 and c_2 represent the coin denominations.

While attempting to solve the Frobenius benchmark, Z3 invokes the `QSAT` solver [12], which models the satisfiability check as a two-player game between existential and universal variables. For instance, in a formula of the form $\exists x \forall y.F$, the existential player seeks to make F true, while the universal player attempts to falsify it by choosing a counter assignment. In its default configuration, QSAT does not eliminate the quantifiers first, but instead checks satisfiability directly through this game-based reasoning. However, QSAT also supports quantifier elimination (qelim) for theories such as Linear Integer Arithmetic (LIA), which admits quantifier elimination, that is, for every formula F, there exists an equivalent quantifier-free formula F' over the same theory.

Support for quantifier elimination in Z3 is available through tactics such as `qe_rec`, which modifies the behavior of QSAT to produce quantifier-free equivalents. While Z3 provides ohter tactics for quantifier elimination, namely `qe` and `qe2`, our focus is on `qe_rec` due to its compatibility with the Frobenius benchmark and its reliable behavior across all tested instances. Other tactics were excluded from this study, as the solver using them occasionally returned `unsat`. Understanding these differences is left to future work. Nevertheless,

Z3 avoids full quantifier elimination by default, as it is often prohibitively expensive in practice [16]. In the case of the Frobenius Coin Problem, however, the formula is restricted to the two coin case, leading to bounded quantifier alternation and a limited number of variables per quantifier block. This structural simplicity, in theory, weakens the conventional justification for avoiding quantifier elimination and makes it a promising candidate for tactic-based solving.

Interestingly, prior empirical evidence suggests that QSAT performs well on large quantified LIA formulas. As shown in [12], QSAT solves 64 benchmark formulas from [17] ranging between 69 to 768 quantifiers with SMT-LIB file sizes between 50KB and 500KB in under 0.08 seconds. In contrast, on the Frobenius benchmark, which is structurally much simpler, QSAT times out on the majority of instances. This unexpected underperformance suggests that Z3's heuristics may be poorly tuned for the Frobenius problem's logical structure. By comparing Z3's default QSAT configuration with a quantifier elimination-based configuration, we aim to uncover whether alternative heuristics, enabled through tactic customization, can better exploit the problem's structure and yield performance improvements.

1.2 Research Question

Motivated by the structural simplicity of the Frobenius Coin problem and the possibility of performance improvement of qelim through the use of tactics, we investigate whether Z3's qelim procedures can improve solver performance compared to its default quantifier handling.

Thus, we formulate the central research question as follows:

> *Does quantifier elimination improve Z3's performance on two-coin Frobenius Coin Problem instances over LIA, compared to its default quantifier handling strategy?*

We evaluate performance on 54 satisfiable Frobenius Coin instances based on consecutive prime pairs, following the setup in [13]. Each instance is encoded via Z3's Python API and tested under two configurations: the **baseline**, using QSAT in default mode, and a **qelim**-based approach using `qe_rec` followed by the **smt** solver. We assess runtime, memory usage, model correctness, and robustness to increasing instance complexity.

We further decompose the research question into the following subquestions:

1. **Correctness**: Do both configurations return satisfiable results with the correct Frobenius number as the model?
2. **Runtime Performance**: How does runtime compare between the default and QE-based approaches as the instance difficulty increases?
3. **Memory Usage**: How does memory consumption vary between configurations under increasing problem size?
4. **Scalability**: At what instance size does the default strategy begin to degrade in comparison to the QE-based approach?

In the remainder of this paper, we first provide essential background on Z3's tactic system and the complexity of Presburger Arithmetic in Section 2. We then describe our methodology for benchmarking Z3 on Frobenius instances in Section 3, followed by the experimental setup in Section 4. Section 5 presents and analyzes the results, and Section 6

offers our conclusions. Finally, we discuss limitations and future work in Section 7 and reflect on the responsible research aspects in Section 8.

2 Background and Motivation

2.1 Tactic Customization and Solver Adaptability

Although Z3 provides a powerful default configuration for solving quantified formulas, its performance can vary significantly across problem domains. This is due to the reliance on solver-internal heuristics, which, while highly optimized for common benchmarks, may fail to generalize to structurally distinct problems [18]. Rather than modifying solver internals, one promising alternative is to customize the tactic pipeline, enabling us to control Z3's strategy.

A good example of this approach is Z3-Alpha [19], a solver based on Z3, which employs reinforcement learning to adaptively compose tactics based on the structure of the problem. Z3-Alpha dynamically selects and schedules tactic applications, achieving state-of-the-art results int SMT-COMP 2024 [20], particularly the Single Query Track for QF_LIA, where it outperformed solvers such as CVC5 [21], Yices2 [22], and SMTInterpol [23] in most satisfiable instances solved.

2.2 Complexity of Presburger Arithmetic and Frobenius Structure

Quantifier elimination (qelim) refers to transforming a quantified formula F into an equivalent quantifier-free formula F' in the same theory. LIA admits quantifier elimination, but exact procedures are known to have high computational cost. Weispfenning established a tight worst-case complexity bound: $2^{O\left(|\Phi|^{(4j)^k}\right)}$, where $|\Phi|$ is the formula size, j is the maximum number of variables per quantifier block, and k is the number of such blocks [24]. For the Frobenius formula (Equation 1), we have $k = 3$, $j = 3$, and we could assume $|\Phi| \approx 80$. Substituting into the bound yields: $2^{O\left(80^{(4\cdot 3)^3}\right)} = 2^{O(80^{1728})}$, which is astronomically large and would make qelim impractical.

However, this worst-case analysis overlooks structural properties that significantly impact practical complexity when reasoning about Presburger Arithmetic. If taken as a whole, PA is decidable in $2^{2^{n^{O(1)}}}$ time by a nondeterministic alternating Turing machine with unbounded space [25].However when restricting the number of quantifier alternations and the number of variables per quantifier block, new bounds for complexity can be established. These are captured more precisely in the Σ_k/Π_k hierarchy. A formula is in the Σ_k class if it has the form:

$$\exists \bar{x}_1 \forall \bar{x}_2 \cdots Q_k \bar{x}_k.\ \varphi_{\text{qf}},$$

where $Q_k \in \{\exists, \forall\}$ and quantifiers alternate k times, and φ_{qf} is a quantifier-free formula. A dual form starting with a universal quantifier belongs to Π_k. The subclass $\text{PA}(i, j)$ contains formulas in Σ_i where each quantifier block binds at most j variables.

For example, a formula like $\exists x.\ \forall y.\ \varphi(x, y)$, where φ is quantifier-free, belongs to $\text{PA}(2, 2)$. Importantly, for any fixed j, the class $\text{PA}(1, j)$ with a single existential quantifier and bounded variable belongs to the P complexity class, meaning it can be decided in polynomial time by a deterministic Turing machine [25]. This difference in complexity highlights that quantifier elimination may be feasible in fragments of PA. More generally, its feasibility is closely tied to the logic's decidability [26].

While the Frobenius problem is generally classified in the Π_2 fragment due to its inherent quantifier structure [27], our focus on the two-coin case imposes bounded quantifier alternation and a restricted number of variables per quantifier block. These structural constraints mitigate the conditions that typically lead to the worst-case behavior of quantifier elimination [24], making a tactic-based approach such as `qe_rec` a good candidate for the Frobenius instances with 2 coins.

3 Methodology

3.1 Instance Generation and Solver Configuration

Each benchmark instance corresponds to a pair of prime coin denominations c_1 and c_2, which define a specific Frobenius problem. We construct the logical encoding in Z3's Python API using the following routine:

```
def build_formula(c1, c2):
    x0, x1, R, P = Ints("x0 x1 R P")
    return And(
        P >= 0,
        ForAll([x0, x1], Implies(And(x0 >= 0, x1 >= 0), P != c1 * x0 + c2 * x1)),
        ForAll([R], Implies(
            ForAll([x0, x1], Implies(And(x0 >= 0, x1 >= 0), R != c1 * x0 + c2 * x1))
    ,
            R <= P
        ))
    )
```

Listing 3: Formula construction for a given coin pair (c_1, c_2)

Depending on the configuration being tested, we apply different solver construction strategies. For the baseline approach, we instantiate Z3 with its default configuration:

```
solver = Solver()
```

Listing 4: Z3 default (baseline) solver instantiation

For the tactic-based approach using quantifier elimination, we construct a composite tactic pipeline using `Then` and pass in tactic-specific parameters, as shown below:

```
p = ParamsRef()
p.set("elim_and", True)
solver = Then(WithParams("simplify", p), "qe_rec", "smt").solver()
```

Listing 5: Quantifier elimination solver using `qe_rec`

We apply the selected approach to the formula produced by `build_formula`, and solving proceeds with a timeout of 60 seconds per run.

3.2 Z3's Approach

Z3's default solving pipeline for $\vec{c} = (2, 3)$ follows a sequence of tactic applications and solver invocations, visualized step-by-step in Figure 2. The process consists of three stages: `simplifier`, `qe-lite`, and `qsat`.

Both `simplifier` and `qe-lite` serve as preprocessing steps that aim to expose the formula's logical structure. As noted by de Moura and Bjørner [28], Z3 begins by applying a repository of simplification rules, followed by a phase known as internalization, where formulas are converted into a normalized form suitable for efficient proof search. While the formula resulting from `simplifier` and `qe-lite` is not in a normal form, these transformations still serve to expose its logical structure and facilitate more effective reasoning during solving.

The pipeline begins with the `simplifier` step, which rewrites the initial formula into a logically equivalent, but structurally simplified form. Implications are rewritten as disjunctions using the equivalence $A \Rightarrow B \rightsquigarrow \neg A \vee B$, and disequalities such as $a \neq b$ are normalized as $\neg(a = b)$. This transformation is illustrated in the transition from the "Input Formula to the "After simplifier" node in Figure 2.

Following simplification, Z3 applies `qe-lite`, a pre-processing tactic that attempts to eliminate quantifiers cheaply. However, as noted in Figure 2, `qe-lite` performs no quantifier reduction in this specific case, instead just normalizing logical structures, namely pushing negations inward.

Finally, the formula is passed to `qsat`, Z3's default solver for quantified SMT formulas in LIA. Rather than eliminating quantifiers upfront, `qsat` models satisfiability as a symbolic two-player game between existential and universal variables. This game-based reasoning enables `qsat` to handle quantified formulas directly. In our example, `qsat` concludes that the formula is satisfiable and returns the model $P = 1$, indicating a solution to the Frobenius instance. This output is shown at the bottom of Figure 2.

3.3 Our Approach

Our alternative tactic sequence, shown in Figure 3, is designed to replicate the pre-processing behavior of Z3's default pipeline before introducing quantifier elimination.

We apply the composite tactic `Then(simplify, qe_rec, smt)`, beginning with the `simplify` tactic using the parameter `elim_and=true` to flatten conjunctions. This ensures that the formula undergoes the same initial structural normalization as in the default configuration, where the combination of `simplifier` and `qe-lite` prepares it for solving. The resulting formula structure matches that produced by Z3 prior to invoking `qsat`, allowing for a fair comparison between default and tactic-based quantifier elimination.

Next, we pass the formula to the `qe_rec` tactic, which tells Z3's internal QSAT solver to perform quantifier elimination, constructing a quantifier-free equivalent of the formula. As illustrated in Figure 3, this transformation introduces explicit arithmetic conditions such as bounds and modular constraints that preserve the semantics of the original quantified formula.

We then pass the quantifier-free formula produced by `qe_rec` to the `smt` tactic, which invokes Z3's core SMT solver. The solver evaluates the arithmetic constraints and returns a satisfiable assignment. In this case, the model returned is again $P = 1$, as shown in the final block of Figure 3.

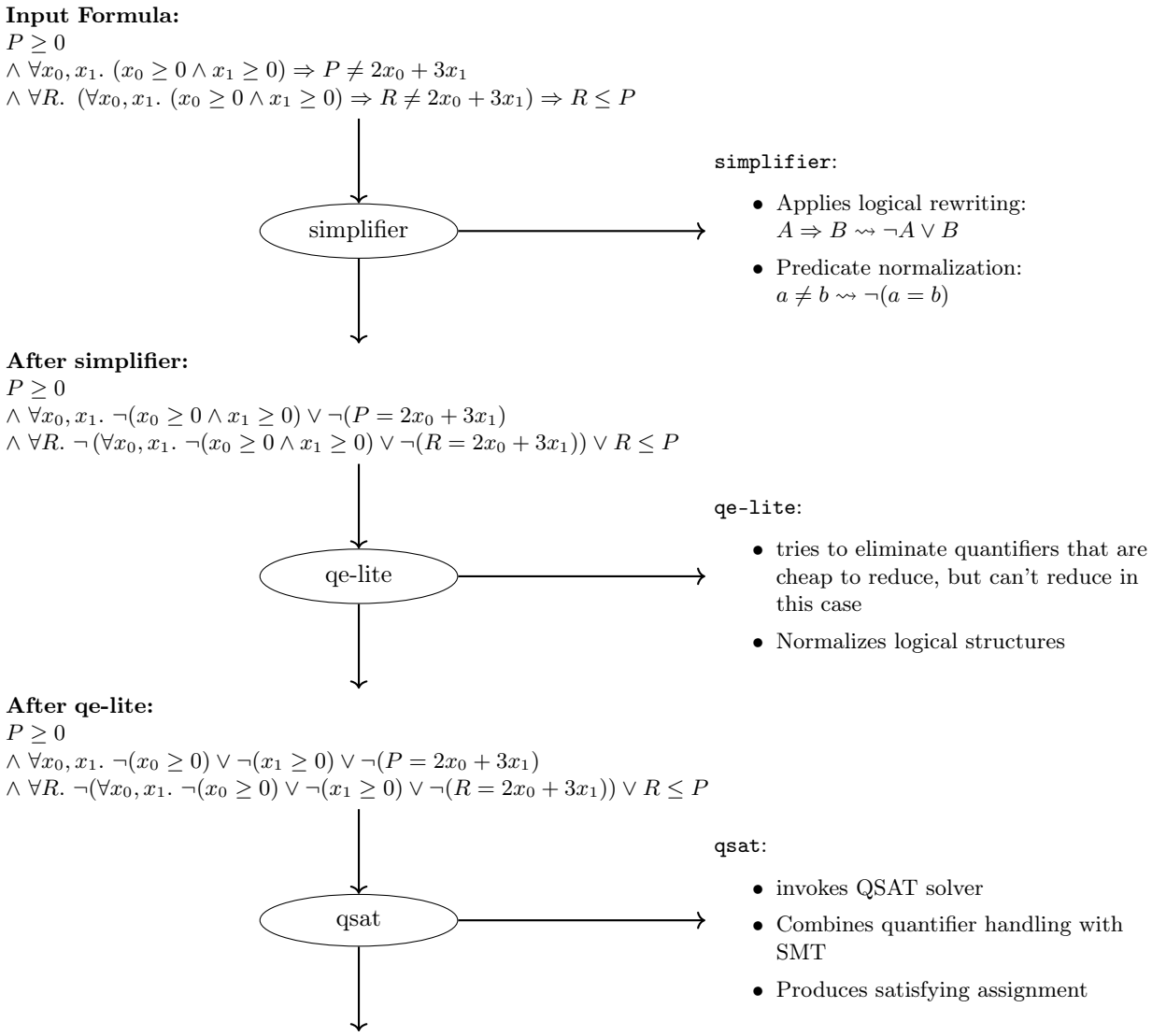


Figure 2: Z3's default solving pipeline for Frobenius instance $\vec{c} = (2, 3)$ using internal modules: simplifier, qe-lite, and qsat. Transformations are shown step-by-step with intermediate formulas and descriptive annotations.

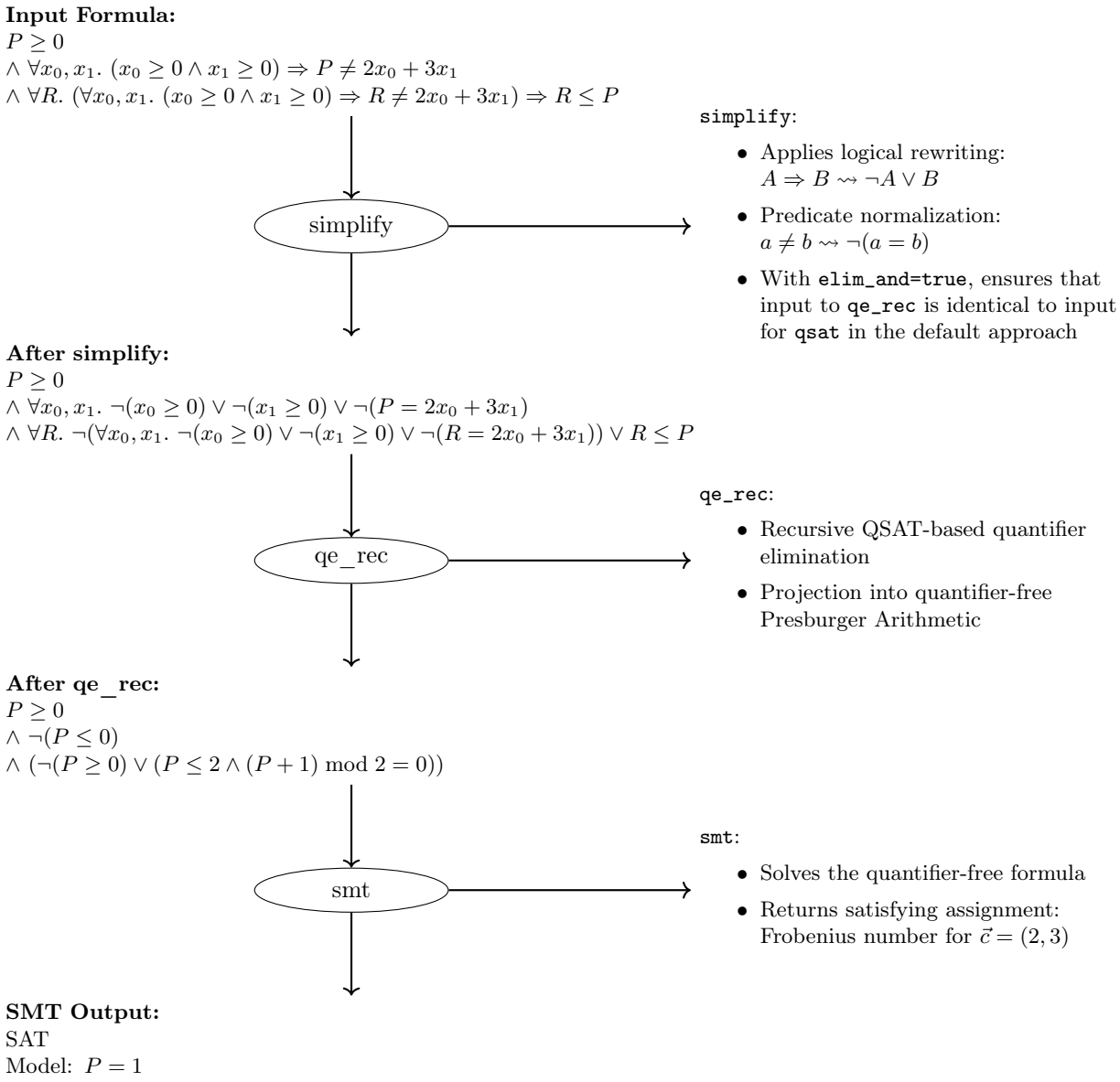


Figure 3: Step-by-step transformation of the Frobenius instance with coin set $\vec{c} = (2, 3)$ using `Then(simplify, "qe_rec", "smt")`, where `simplify` is the `simplify` tactic applied with the parameter `elim_and = true`. The diagram shows logically equivalent formula transformations and tactic-level annotations.

4 Experimental Setup

All experiments were implemented using Z3's Python API, with each solver run executed in a separate subprocess to ensure clean state and accurate resource measurement. Formulas were generated using a dedicated function (Listing 3) that encodes the Frobenius problem as a quantified LIA formula. To test each configuration, we instantiated a new solver per run and applied a 60-second timeout.

We compared two configurations: the default Z3 strategy using `qsat`, and a quantifier elimination approach using `Then(simplify, "qe_rec", "smt")` with the `elim_and=true` parameter. Each was evaluated on 54 satisfiable Frobenius instances, each defined by a distinct pair of consecutive prime coin denominations (e.g., (3, 5), (5, 7), etc.), with 30 independent runs per instance.

For each run, we recorded the coin pair, solver result (`sat`, `unsat`, or `unknown`), the satisfying model (if available), total execution time, and peak memory usage. The latter two metrics were extracted from the `solver.statistics()` object. These were used to evaluate correctness, runtime performance, memory efficiency, and scalability as problem size increased.

The full codebase and configuration files are available at: `https://github.com/PaulAnton03/z3-research-project.git`.

5 Results

Both approaches returned the Frobenius number on solved instances, with no `unsat` returned by any of them. Our approach was able to consistently solve one more instance of the Frobenius coin problem. Below is a table summarizing the results for number of instances solved and runtime performance on solved instances.

Coin 1	#solved QSAT	#solved QE_rec	Mean QSAT (s)	RSE QSAT (%)	Mean QE_rec (s)	RSE QE_rec (%)
2	30	30	0.02	0.0%	0.02	0.0%
3	30	30	0.04	0.0%	0.03	0.0%
5	30	30	0.12	8.3%	0.06	0.0%
7	30	30	0.68	10.3%	0.13	0.0%
11	30	30	1.98	5.1%	0.68	4.4%
13	30	30	15.37	8.5%	1.44	4.9%
17	**20**	30	28.28	**14.8%**	6.37	4.9%
19	**26**	30	36.87	7.1%	10.13	3.7%
23	**0**	**30**	-	-	20.85	4.1%
> 23	0	0	-	-	-	-

Table 1: Comparison of QSAT and QE_rec on the first 9 Frobenius instances. RSE is the relative standard error, computed as $\frac{\text{SE}}{\text{mean}} \times 100$.

As shown in Table 1, both configurations solved the first six Frobenius instances consistently. However, as the problem complexity increased, differences began to emerge. Our approach using the tactic `qe_rec` successfully solved all nine of the selected instances, while the default `qsat` strategy failed on the final instance $\vec{c} = (23, 29)$ and partially failed on two others $\vec{c} = (17, 19)$, $\vec{c} = (19, 23)$. In total, across the full benchmark of 54 instances, `qe_rec` was able to solve one more instance than the baseline approach.

We report runtimes as arithmetic means over 30 independent runs per instance. We use the mean to estimate the expected runtime of each solver configuration, providing a reliable basis for performance comparison across repeated trials. This metric captures the

central tendency of runtime behavior and supports statistical inference. For instances where `qsat` failed to complete all 30 runs, we assigned the timeout threshold of 60 seconds to each unresolved case. As a result, the reported mean runtimes for $\vec{c} = (17, 19)$ and $\vec{c} = (19, 23)$ under `qsat` are likely underestimated.

To quantify the confidence in these sample means, we report the *relative standard error* (RSE), defined as: $\text{RSE} = \frac{\text{SE}}{\bar{x}} \times 100\%$, where $\text{SE} = \frac{s}{\sqrt{n}}$, s is the sample standard deviation, $\bar{x}$ is the sample mean, and $n = 30$ is the number of runs. RSE expresses the variability in the sample mean as a percentage of the mean itself. A higher RSE indicates less reliability in the reported mean.

In contrast to the `qsat` configuration, our approach using `qe_rec` not only solved all nine instances but did so with greater consistency. The RSE values for `qe_rec` remained below 5% for most instances and did not exceed 9% in any case. This indicates that the observed mean runtimes are likely close to the true expected performance of the tactic, even on more difficult instances.

To better visualize the performance differences between the two configurations, we compute the per-instance speed-up of `qe_rec` relative to the baseline. This speed-up is defined as the ratio of the mean runtime of `qsat` to that of `qe_rec`, and quantifies how many times faster our approach was, on average, for each instance.

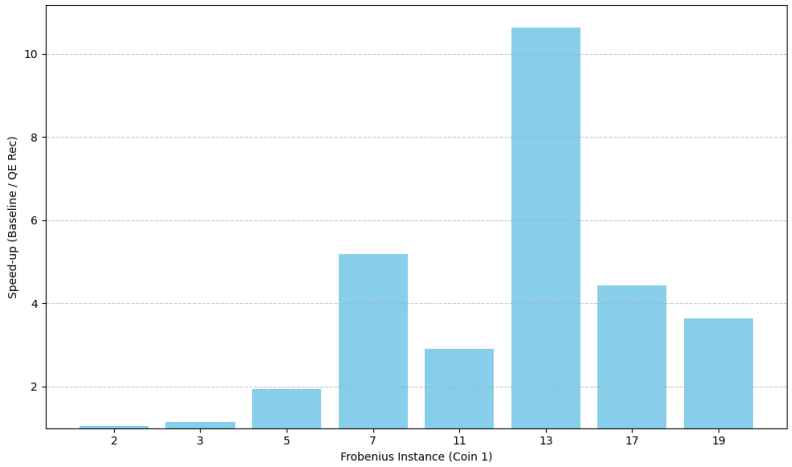


Figure 4: Per-instance speed-up of `qe_rec` over the baseline `qsat` tactic, computed as the ratio of mean runtimes.

Figure 4 illustrates the relative performance advantage of our approach. A value of x on the vertical axis indicates that `qe_rec` was, on average, x times faster than the baseline on that instance.

The plot shows that `qe_rec` consistently outperforms `qsat`, with the speed-up increasing significantly as instance complexity grows. For instance, in the case of $\vec{c} = (13, 17)$, the speed-up exceeds $10\times$, meaning that `qe_rec` completed in less than one-tenth the time required by the baseline. Even for smaller instances, modest but consistent speed-ups are observed, typically around $1.2\times$ to $5\times$.

Figure 5 shows the memory usage trends for both solver configurations across all 54 Frobenius instances, plotted against the value of the first coin in each pair. Each curve depicts the mean peak memory consumption in megabytes, which we computed over 30

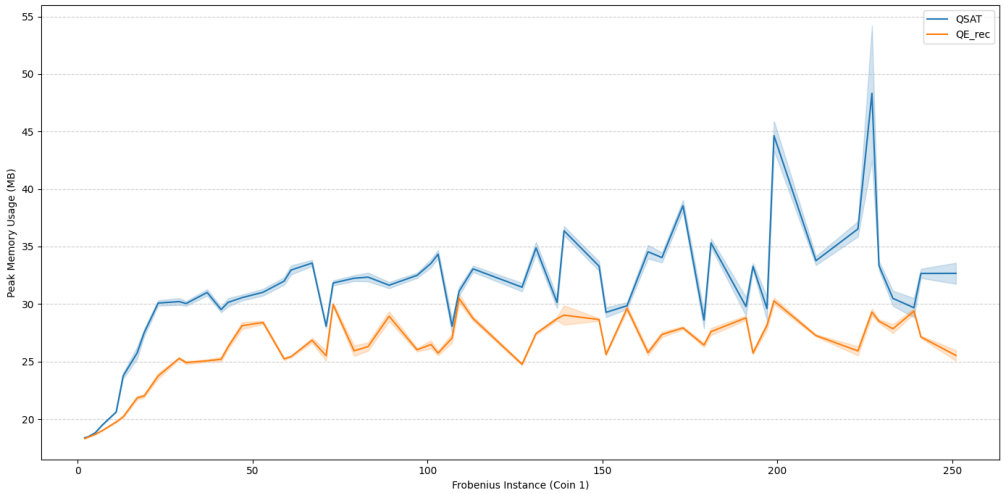


Figure 5: Peak memory usage across 54 Frobenius instances for both `qsat` and `qe_rec`. Each line shows the mean memory usage over 30 runs per instance. The shaded regions represent the standard error of the mean (SE).

independent runs per instance. To illustrate variability, we shaded the regions around each curve to represent the standard error of the mean (SE). We chose SE because it reflects the precision of the sample mean as an estimate of the true population mean, allowing us to assess the consistency of memory usage across runs. Although runtime performance remains a key metric, solver efficiency also depends on memory usage, especially in more complex instances where limited resources may lead to failure. To complement our runtime analysis, we now compare peak memory consumption.

Overall, we observe that `qe_rec` exhibits lower and more stable memory usage than `qsat`. This advantage becomes especially clear as instance complexity increases, where `qsat` starts to show sharp spikes and greater variability, with memory usage consistently exceeding 30 MB. In contrast, `qe_rec` maintains memory usage within a narrow band, between 25 and 30 MB across nearly all instances, and SE remains consistently low. These results indicate that `qe_rec` not only consumes less memory but does so more predictably.

6 Conclusion

In this work, we investigated whether applying quantifier elimination improves Z3's performance on the Frobenius Coin Problem, a benchmark of quantified formulas over linear integer arithmetic (LIA). This was motivated by two key factors: first, Z3's tactic pipeline allows for customization to better accommodate structurally distinct problems, offering an alternative to fixed heuristic strategies. Second, the two-coin Frobenius encoding involves bounded quantifier alternation and a limited number of variables per block, which suggests that quantifier elimination may be computationally feasible. To test this, we applied the `qe_rec` tactic to eliminate quantifiers prior to solving.

We framed our central research question as follows: *Does quantifier elimination improve Z3's performance on two-coin Frobenius Coin Problem instances over LIA, compared to its*

default quantifier handling strategy? To explore this, we examined four subquestions:

1. **Correctness:** Do both configurations return satisfiable results with the correct Frobenius number as the model?
2. **Runtime Performance:** How does runtime compare between the default and QE-based approaches as the instance difficulty increases?
3. **Memory Usage:** How does memory consumption vary between configurations under increasing problem size?
4. **Scalability:** At what instance size does the default strategy begin to degrade in comparison to the QE-based approach?

Our evaluation produced the following results:

1. **Correctness:** Both configurations returned `sat` and valid models on all instances they successfully solved.
2. **Runtime Performance:** The `qe_rec` tactic consistently achieved faster runtimes of up to 10× faster than `qsat` with improvements becoming more pronounced as the problem size increased.
3. **Memory Usage:** The `qe_rec` configuration consumed less memory and showed more consistent usage patterns across instances, with narrow standard error margins.
4. **Scalability:** The `qsat` configuration began to fail on instances starting with $\vec{c} = (17, 19)$ and failed completely on $\vec{c} = (23, 29)$. In contrast, `qe_rec` successfully solved all 30 runs for $\vec{c} = (23, 29)$.

These results confirm that quantifier elimination using `qe_rec` improves both runtime and memory efficiency over Z3's default QSAT-based strategy on the Frobenius Coin Problem instances tested and also highlight how customizing Z3's approach, which is based on hand-crafted heuristics, can lead to overall better performance. Nevertheless, this does not imply that our approach is superior in general, as performance may vary on larger instances, an aspect which was not covered due to the limitations imposed by the 60 second timeout and number of instances being included in the experiments.

7 Limitations and Future Work

This study focused solely on two-coin, satisfiable Frobenius instances, where the quantifier structure is relatively simple. Future work could extend the analysis to instances involving three or more coins, as well as unsatisfiable cases, to evaluate how `qe_rec` scales with increased quantifier complexity. While the improved performance of our quantifier elimination approach suggests a structural advantage, this work did not fully investigate how Z3 internally processes these instances. Further analysis is needed to determine whether the observed improvements are directly attributable to the simplified quantifier structure or arise from other factors such as solver heuristics or internal optimizations.

Support for quantifier elimination in Z3 is available through multiple tactics, including `qe`, `qe2`, and `qe_rec`. This study focused exclusively on `qe_rec` due to its consistent and reliable behavior on the Frobenius benchmark. Other tactics were excluded because they

occasionally produced incorrect results, namely returning `unsat` for satisfiable instances. Future work should investigate these tactic differences in greater depth to understand the reasons behind the failures, and possibly address them.

Comparing Z3 with other SMT solvers that support quantifier elimination for LIA, such as CVC5, could provide additional context. Finally, learning-based methods, such as reinforcement learning approaches used in Z3-Alpha, could be leveraged to enable automatic and adaptive tactic scheduling based on formula structure.

8 Responsible Research

This section reflects on the integrity, transparency, and potential impact of our research. We address reproducibility, data accessibility, and ethical considerations in line with established scientific best practices.

8.1 Reproducibility

Our research complies with the Netherlands Code of Conduct for Research Integrity (2018) and aligns with the FAIR principles for responsible data management [29]. To support reproducibility, we make our entire codebase publicly available via a GitHub repository[1], including all scripts for generating benchmarks, configuring solver tactics, and collecting results.

All tools used in our study are open source. In particular, we rely on Z3 version 4.14.1 (64-bit) and explicitly document all tactic parameters and timeout settings. Our benchmark generator is deterministic and produces instances based on specified prime inputs, ensuring that others can reconstruct the same test suite.

We log results in machine-readable formats (CSV) and provide summary statistics to support analysis and replication. No specialized hardware is required. All experiments were run on standard CPUs, and performance variability due to solver nondeterminism is mitigated by repeated trials and averaged metrics.

8.2 Ethical Impact

This work does not involve personal data, automated decision-making, or human participants. Its contributions focus solely on solver configuration and performance evaluation for quantified formulas in linear integer arithmetic.

Nevertheless, as SMT solvers are increasingly used in sensitive domains such as program verification, cybersecurity, and decision systems, we advise caution when applying tactic-based configurations in contexts requiring formal guarantees. It is essential to verify that any simplifications or transformations introduced by tactics preserve correctness, especially when used in critical applications.

References

[1] N. Rungta, "A billion smt queries a day," 2022. [Online]. Available: https://www.amazon.science/publications/a-billion-smt-queries-a-day

[1] `https://github.com/PaulAnton03/z3-research-project.git`

[2] T. N. Alyahya, M. E. B. Menai, and H. Mathkour, "On the structure of the boolean satisfiability problem: A survey," *ACM Comput. Surv.*, vol. 55, no. 3, Mar. 2022. [Online]. Available: https://doi.org/10.1145/3491210

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

[4] K. R. M. Leino, "This is boogie 2," *manuscript KRML*, vol. 178, no. 131, p. 9, 2008.

[5] P. Müller, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," in *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583*, ser. VMCAI 2016. Berlin, Heidelberg: Springer-Verlag, 2016, p. 41â62. [Online]. Available: https://doi.org/10.1007/978-3-662-49122-5_2

[6] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *International conference on logic for programming artificial intelligence and reasoning*. Springer, 2010, pp. 348–370.

[7] L. de Moura and N. Bjørner, "Z3 internals," https://z3prover.github.io/papers/z3internals.html, 2024, accessed: 10th June 2025.

[8] C. Barrett, A. Stump, C. Tinelli *et al.*, "The smt-lib standard: Version 2.0," in *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, vol. 13, 2010, p. 14.

[9] C. Haase, "A survival guide to presburger arithmetic," *ACM SIGLOG News*, vol. 5, no. 3, p. 67â82, Jul. 2018. [Online]. Available: https://doi.org/10.1145/3242953.3242964

[10] H. Ding, Z. Wang, Y. Yang, D. Zhang, Z. Xu, H. Chen, R. Piskac, and J. Li, "Proving query equivalence using linear integer arithmetic," *Proceedings of the ACM on Management of Data*, vol. 1, no. 4, pp. 1–26, 2023.

[11] A. Reynolds, C. Tinelli, A. Goel, S. Krstić, M. Deters, and C. Barrett, "Quantifier instantiation techniques for finite model finding in smt," in *Automated Deduction – CADE-24*, M. P. Bonacina, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 377–391.

[12] N. S. Bjørner and M. Janota, "Playing with quantified satisfaction." *LPAR (short papers)*, vol. 35, pp. 15–27, 2015.

[13] M. Hečko, "Rozhodování logiky pomocí automatů [online]," Master's thesis, Brno University of TechnologyBrno, 2024 [cit. 2025-05-23], sUPERVISOR:. [Online]. Available: https://theses.cz/id/yt82t6/

[14] J. L. Ramírez Alfonsín, *The Diophantine Frobenius Problem*. Oxford University Press, 12 2005. [Online]. Available: https://doi.org/10.1093/acprof:oso/9780198568209.001.0001

[15] P. Rümmer, "A constraint sequent calculus for first-order logic with linear integer arithmetic," in *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 2008, pp. 274–289.

[16] A. Reynolds, T. King, and V. Kuncak, "Solving quantified linear arithmetic by counterexample-guided instantiation," *Formal Methods in System Design*, vol. 51, pp. 500–532, 2017.

[17] A.-D. Phan, N. Bjørner, and D. Monniaux, "Anatomy of alternating quantifier satisfiability (work in progress)," in *10th International Workshop on Satisfiability Modulo Theories*, 2012, p. 6.

[18] L. De Moura and G. O. Passmore, "The strategy challenge in smt solving," in *Automated Reasoning and Mathematics: Essays in Memory of William W. McCune*. Springer, 2013, pp. 15–44.

[19] Z. Lu, S. Siemer, P. Jha, F. Manea, J. Day, and V. Ganesh, "Z3-alpha: a reinforcement learning guided smt solver," *System Description: SMT-COMP*, 2023.

[20] F. Bobot, C. List, M. Bromberger, and M. Jonáš, "19th international satisfiability modulo theories competition (smt-comp 2024): Rules and procedures."

[21] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli *et al.*, "cvc5: A versatile and industrial-strength smt solver," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2022, pp. 415–442.

[22] B. Dutertre, "Yices 2.2," in *Computer Aided Verification*, A. Biere and R. Bloem, Eds. Cham: Springer International Publishing, 2014, pp. 737–744.

[23] J. Christ, J. Hoenicke, and A. Nutz, "Smtinterpol: An interpolating smt solver," in *International SPIN Workshop on Model Checking of Software*. Springer, 2012, pp. 248–254.

[24] D. Chistikov and C. Haase, "On the complexity of quantified integer programming," in *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*, vol. 80. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, p. 94.

[25] C. Haase, "Subclasses of presburger arithmetic and the weak exp hierarchy," in *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2014, pp. 1–10.

[26] B. Schüpp, "Quantifier elimination," 2021.

[27] D. Chistikov, "An introduction to the theory of linear integer arithmetic," in *44th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2024)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024, pp. 1–1.

[28] L. M. de Moura and N. S. Bjørner, "Proofs and refutations, and z3." in *LPAR Workshops*, vol. 418. Doha, Qatar, 2008, pp. 123–132.

[29] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L. B. da Silva Santos, P. E. Bourne *et al.*, "The fair guiding principles for scientific data management and stewardship," *Scientific data*, vol. 3, no. 1, pp. 1–9, 2016.