Theory of Hybrid Systems Informatik 2
RWTH AACHEN UNIVERSITY

The present work was submitted to the LuFG Theory of Hybrid Systems

MASTER OF SCIENCE THESIS

IMPLEMENTATION OF CYLINDRICAL ALGEBRAIC COVERINGS FOR QUANTIFIER ELIMINATION

Philip Kroll

Examiners:
Prof. Dr. Erika Ábrahám
Prof. Dr. Jürgen Giesl

Additional Advisor:
Jasper Nalbach

Aachen, 6. December 2023

Abstract

The *Cylindrical Algebraic Coverings (CAlC)* Algorithm was originally developed for solving the satisfiablity of conjunctions for the theory of *Non-linear Real Arithmetic (NRA)* and implemented in the SMT-Solvers *SMT-RAT* and *CVC5*. Later, it was generalized to solve the *Quantifier Elimination (QE) Problem* and general formulas directly, but not implemented yet. The QE problem is concerned with eliminating all quantifiers, the universal quantifier $\forall$ and the existential quantifier $\exists$, from a first-order logic formula and obtaining an equivalent formula as the result. Further, the CAlC algorithm can be used as a decision procedure in the context of *Satisfiability Modulo Theories (SMT)*.

In this thesis, we present an implementation of the CAlC algorithm for quantifiers in the SMT solver *SMT-RAT* suitable to produce a result for both the QE problem and the decision problem for a general formula of the theory of NRA. We also present a novel approach to split the formula of interest into multiple different formulas that can be solved independently and whose results can be combined to form the solution of the original formula. Additionally, we present three different variable ordering heuristics that can be used to improve the performance of the CAlC algorithm. Finally, we compare the performance of our implementation to other programs that can solve the QE problem and the decision problem for the theory of NRA.

Erklärung

Hiermit versichere ich, dass ich die vorgelegte Arbeit selbstständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe. Alle benutzten Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate wurden als solche gekennzeichnet.

Philip Kroll
Aachen, den 18. December 2023

Contents

Chapter 1

Introduction

Few technologies have had as profound of an impact on human life as modern computer systems. Algorithms, which are systematic procedures for processing information, have become increasingly crucial in modern society. They play an essential role in every aspect of life. These algorithms are generally designed to solve specific problems and are highly efficient. However, guaranteeing the correctness of their solutions is a significant challenge and can be just as important as their speed. As a result, general frameworks have been established, allowing many problems to be described without requiring the user to have any previous problem-solving knowledge. This task is delegated to a solver, which assists in checking the correctness of problem-specific algorithms and enables them to resolve logical problems through deduction. The *Boolean Satisfiability Problem (SAT)* is a well-established and extensively researched problem that was the first to be proven NP-complete [Coo71]. This means that a wide range of problems in computer science can be reduced to this problem. The problem statement of SAT has been extended for first-order logic, namely *Satisfiability Modulo Theories (SMT)* [BT18]. SMT formulas are Boolean combinations of constraints from some background theory. The theory of interest in this thesis is *Non-linear Real Arithmetic (NRA)*. In this theory, variables may be universally or existentially quantified, allowing for the creation of intricate formulas. Further, the theory of non-linear real arithmetic has been shown to admit quantifier elimination [Tar51]. This means that any formula of this theory can be transformed into an equivalent quantifier-free formula. It is known that the complexity of real quantifier elimination is doubly exponential in the number of quantifier alternations [DH88]. The first practically used procedure for quantifier elimination is the *Cylindrical Algebraic Decomposition (CAD)* [Col76]. The original algorithm has been improved with many enhancements, including projection methods [McC98, Hon90], partially built CADs [CH91] and efficient projection orders [HEW$^+$14, DSS04]. Based on CAD, specifically for the existential fragment of the theory, a conflict-driven approach, namely the *Cylindrical Algebraic Covering (CAlC)* algorithm, has been developed [ÁDEK21]. Lately, this conflict-driven approach has been extended for deciding the satisfiability of NRA formulas and for real quantifier elimination [KN22]. This thesis will present this approach in detail and a novel adaptation. This adaptation includes that the formula to solve may be split into two or more subformulas given particular circumstances. These subformulas can then be solved independently, giving the result for the original formula. Further, this algorithm and its adaptation is implemented and the performance for

both the decision problem and the quantifier elimination problem is evaluated and compared with the results of other state-of-the-art solvers.

Chapter 2

Preliminaries

This chapter is an introduction to the theoretical knowledge required for the central part of this thesis. It aims to clarify the fundamental concepts and principles that form the basis of the algorithms presented in the following chapters. Firstly, we will provide an explanation of the problem setting as well as the theory of interest. We will introduce both the decision problem and the quantifier elimination problem for non-linear real arithmetic. Additionally, we will briefly discuss the cylindrical algebraic coverings method for solving the existential fragment of non-linear real arithmetic. Furthermore, we will introduce several concepts necessary for the main part of this thesis.

2.1 Boolean Satisfiability Problem

The *Boolean Satisfiability Problem (SAT)* is the problem of determining if a given formula in propositional logic is satisfiable. A *propositional logic formula* is built by a fixed set of *atomic propositions* to which the value of the Boolean *constants* $\{0, 1\}$ can be assigned. A propositional logic formula can then be constructed with atomic propositions and the standard logical connectives with the usual semantics.

Definition 2.1.1 (Propositional Logic Formula)**.**

$$\varphi := x \quad | \quad \neg x \quad | \quad \varphi \wedge \varphi \quad | \quad \varphi \vee \varphi \quad | \quad \varphi \oplus \varphi \quad | \quad \varphi \implies \varphi \quad | \quad \varphi \iff \varphi$$

where x is an atomic proposition.

A variable assignment assigns Boolean constants to variables in a logic formula, which allows the formula to be evaluated. If a variable assignment exists such that the formula evaluates to *true*, the formula is said to be satisfiable. If no such assignment exists, the formula is unsatisfiable. The concept of satisfiability can also be extended to first-order logic through *Satisfiability Modulo Theories (SMT)* [BT18]. In SMT, a Boolean combination of constraints from some background theory replaces the Boolean combination of atomic propositions. This thesis focuses on the background theory of *Non-linear Real Arithmetic (NRA)*.

2.2 Non-Linear Real Arithmetic

This section introduces the theory of non-linear real arithmetic, also called *NRA*. It is a first-order theory over $\mathbb{R}$, also called the *reals*. The theory enables equalities and inequalities over real numbers.

Definition 2.2.1 (Polynomial Constraint)**.**
A *polynomial constraint* is a multivariate polynomial compared to zero with a comparison predicate:

$$C := \sum_{k=0}^{n} a_k \prod_{j=0}^{m} x_j^{e_{k,j}} \bowtie 0$$

Where:

- $a_k \in \mathbb{Q}$ are the coefficients
- x_i are real-valued variables
- $e_{k,j} \in \mathbb{Z}$ are the exponents
- $\bowtie \in \{\leq, <, =, \neq, >, \geq\}$ is the comparison predicate

Since we do not deal with other theories in this thesis, the term *constraint* will be used synonymously with *polynomial constraint*.

Definition 2.2.2 (NRA Formulas)**.**
Let C be a constraint. A formula in non-linear real arithmetic φ may be of the following form:

$$\varphi := C | \exists x.\varphi | \forall x.\varphi | \varphi \vee \varphi | \varphi \wedge \varphi | \varphi \Rightarrow \varphi | \varphi \Leftrightarrow \varphi | \varphi \oplus \varphi$$

where x is a variable.

Further, we introduce the notation $x_i \in \texttt{vars}(\varphi)$ if the variable x_i appears in any constraints used in φ. If a variable is not specifically quantified, either existentially or universally, it is denoted as a *free* variable. A formula φ is said to be *quantifier-free* if for every variable $x \in \texttt{vars}(\varphi)$, x is a free variable.

In this thesis, we are generally interested in a quantified logical expression involving polynomial equalities and inequalities with real-valued variables. In the context of SMT, such an expression is called a *NRA Formula* or *Tarski Formula* [Bro99]. From now on, we will denote such expressions simply as a *formula*.

Definition 2.2.3 (Prenex Normal Form)**.**
A formula φ is in prenex normal form if it has the form:

$$\varphi := Q_{k+1} x_{k+1} \dots Q_n x_n . \bar{\varphi}(x_1, \dots, x_n)$$

with quantifiers $Q_i \in \{\forall, \exists\}$ and $\bar{\varphi}$ is quantifier-free. We call $\bar{\varphi}$ the *matrix* of φ and $Q_{k+1} x_{k+1} \dots Q_n x_n$ the *prefix* of φ.

If it holds that $k \neq 0$, then φ has free variables that are not explicitly quantified. Any given formula can be transformed to prenex normal form by applying

the following rules [Hin18]. All quantifiers are to be the pushed front of the formula while preserving the order of the quantifiers used. Further, one has to ensure that variables are renamed when multiple quantifiers reference them and that the quantifiers are properly changed when pushed over a negation. From now on, we assume that the variables are ordered under the total ordering $\prec$ of their labels, i.e., $\{x_0 \prec x_1 \prec \ldots \prec x_n\}$.

Definition 2.2.4 (Main Variable and Level)**.**

- The main variable of a formula is the highest one in the ordering present in the formula. The main variable of a formula φ is denoted as $\texttt{main}(\varphi)$.
- The level of a polynomial is the position of the main variable in the variable ordering $x_0 \prec x_1 \prec \ldots \prec x_n$.

We denote the constraints in a formula φ by $\texttt{constraints}(\varphi)$. Further, we denote the constraints with the main variable x_i in a formula φ by $\texttt{constraints}_i(\varphi)$. Given a (partial) sample point $s \in \mathbb{R}^k$ we denote the extension of s to $(s_1, \ldots, s_k, s_{k+1}) \in \mathbb{R}^{k+1}$ by $s \times s_{k+1}$.

Definition 2.2.5 ((Partial) Formula Evaluation)**.**
Let φ be a formula of level n. Let $s \in \mathbb{R}^k$ with $k \leq n$ be a (partial) sample point. We denote the (partial) evaluation up to level k of φ over s by $\varphi[s]$. That is, we substitute s for the first k variables in the formula φ. Partial formula evaluation leads to the following cases when evaluating a formula φ:

$\varphi[s] = \texttt{True}$: When substituting s into the $\texttt{constraints}(\varphi)$ the Boolean combination of the resulting constraints is *True*.

$\varphi[s] = \texttt{False}$: When substituting s into the $\texttt{constraints}(\varphi)$ the Boolean combination of the resulting constraints is *False*.

When for a given formula φ and (partial) sample point s it holds that $\varphi[s] \notin \{\texttt{True}, \texttt{False}\}$, we say that the evaluation is *inconclusive*.

Example 2.2.1 ((Partial) formula evaluation)**.**

- $(x_1 \cdot x_2 > 1)[(x_1 = 1, x_2 = 1)] = 1 > 1 = \texttt{False}$
- $(x_1 \cdot x_2 > 1)[(x_1 = 1, x_2 = 2)] = 2 > 1 = \texttt{True}$

Definition 2.2.6 (Quantifier Elimination Problem (QE))**.**
Given a formula φ with some quantified variables, produce a quantifier-free formula equivalent to φ.

It has been shown that the theory of non-linear real arithmetic admits quantifier elimination [Tar51]. In the quantifier elimination problem, the free variables of φ can be understood as *parameters*. The existence of a quantifier-free equivalent is known as the *Tarski-Seidenberg Principle* [Sei54, Tar51]. Tarski gave the first constructive solution, but the complexity of his solution was indescribable, meaning no elementary function could describe it [Tar51]. The quantifier elimination problem has been proven to have a doubly exponential lower bound for complexity in the number of quantifier alternations [DH88]. The first practically viable solution had to await the concept of the *Cylindrical Algebraic Decomposition (CAD)* [Col76]. To produce a solution for

the QE problem for a formula φ consisting of the variables $\{x_1, \ldots, x_n\}$, we need to produce a corresponding CAD of $\mathbb{R}^n$. A CAD is a finite partitioning of $\mathbb{R}^n$ into cells, such that the conjunction of polynomial equalities and inequalities defines each cell. A cell is said to be *sign-invariant* for a set of polynomials if the sign of each polynomial remains consistent over the cell, either positive, negative or zero. In a CAD, each cell is sign-invariant for all polynomials that occur in the formula φ. This implies that the formula φ has the same truth value over each cell, and thus the formula can be evaluated over a representative sample point of each cell. Hence, it is only necessary to consider the truth or falsity of φ at the finite number of sample points and query their algebraic definitions to form the respective quantifier-free output formula. If x_i is existentially quantified, we require the truth of φ for at least one sample point regarding that variable. However, If the variable x_i is universally quantified, we require the truth of φ for all representative sample points regarding that variable. Note that such a CAD is finer than required for real quantifier elimination since, as well as answering the QE question asked, it could also answer another that involved the same polynomials but with possibly different comparison operators and even different quantifiers, as long as the variables are still quantified in the same order. For a more detailed introduction to CADs, we refer to [BDE+16, CJ12].

Definition 2.2.7 (Decision Problem)**.**
Given a formula φ, decide whether this formula is *satisfiable*. That is, a variable assignment exists for the free variables of φ, such that the formula evaluates to `True`. If no such variable assignment exists, the given formula is *unsatisfiable*.

In the following, we may abbreviate the satisfiability of a given formula as `SAT` and the unsatisfiability as `UNSAT`.

Example 2.2.2 (Decision Problem)**.**

- $\varphi := \exists x.x^2 \geq 0$: φ is satisfiable since there exists a value for x such that $x^2 \geq 0$, such as $x := 0$.
- $\varphi := \exists x.x^2 \geq y$, where y is a free variable. Now, φ is satisfiable given any sample point $s = (s_1) \in \mathbb{R}^1$.
- $\varphi := \forall x.x \geq y$: where y is a free variable. This formula is unsatisfiable, as there is no fixed value $s \in \mathbb{R}^1$ such that $x \geq s$ for all possible $x \in \mathbb{R}$.
- $\varphi := (\exists x.x^2 \geq y) \vee (\forall x.x \geq y)$, where y is a free variable. This formula is satisfiable since the first part of the disjunction is satisfiable.

The decision problem is a special case of the quantifier elimination problem, where the output is a binary answer. This results in the fact that any algorithm that solves the quantifier elimination problem can be used to solve the decision problem. If the formula φ has free variables, these can be considered to be implicitly quantified existentially. As the formula φ has no free variables, the only possible equivalent quantifier-free formulas ψ are either `True` or `False`. If ψ is `True`, then φ is satisfiable, otherwise it is unsatisfiable.

2.3 Cylindrical Algebraic Coverings

In the following, we briefly present the idea behind the *Cylindrical Algebraic Coverings(CAlC)* method for checking the satisfiability of formulas of the existential

fragment of non-linear real arithmetic [ÁDEK21]. In the existential fragment, we only allow existentially quantified variables. The fundamental idea is to recursively construct a (partial) sample point and collect intervals that represent unsatisfiable regions above this sample point. When a sample point can not be extended because these unsatisfiable intervals form a covering of the real line in the current dimension, the covering is projected into the prevision dimension to refute the current sample point of that dimension. We then backtrack and choose a different sample value for the sample point of the highest level. Eventually, either a full sample point is constructed such that the given formula of interest evaluates to `True`, and we can conclude satisfiability and return `SAT`, or an unsatisfiable covering of the first dimension is constructed, and we can conclude unsatisfiability and return *UNSAT*. The algorithm starts by constructing unsatisfiable intervals for the first dimension based on univariate polynomials with the main variable x_1 and then tries to select a value s_1 for the variable x_1 outside of these intervals. If such a value exists, the algorithm is recursively called with the extended sample point (s_1). The algorithm continues by constructing unsatisfiable intervals for the second dimension based on polynomials with the main variable x_2. When substituting the sample point (s_1) into the constraints with the main variable x_2, these become univariate and thus suitable for identifying unsatisfiable intervals for x_2. If a value s_2 exists for the variable x_2 outside of these intervals, the algorithm is recursively called, and the procedure continues. This process is continued until either a full sample point is constructed and all constraints are satisfied, and we can return `SAT`, or for some dimension i, there exists no value s_i for the variable x_i outside of the unsatisfiable intervals for x_i. In this case, a set of unsatisfiable intervals for x_i forms a covering of the real line. This covering is generalized by projecting it to dimension $i-1$. The tools for this projection step are taken from the CAD algorithm. The sample point s_{i-1} is refuted and generalized to an unsatisfiable interval for the same reasons. The algorithm then continues by backtracking to the previous dimension and trying to choose a different value for s_{i-1}. This process is continued until either a full sample point is constructed and all constraints are satisfied, and we can return `SAT`, or we obtain a covering for the first dimension and can conclude unsatisfiability and return `UNSAT`.

In contrast to cells created in a CAD, which form a disjoint partitioning of the real space, the intervals created in the CAlC algorithm may overlap. To generalize intervals over a partial sample point, we need to attach algebraic information in the form of sets of polynomials whose *order-invariance* characterizes satisfiability-invariant regions of a formula in multiple dimensions. It holds that order-invariance is a stronger property than sign-invariance and is used in the McCallum projection operator [McC98]. The property of sign-invariance is used in the McCallum projection operator to make the set of polynomials resulting from the projection smaller. These are called *algebraic intervals* and are defined in the following together with what information they contain. A polynomial *vanishes* at a sample point if it evaluates to zero at that sample point.

Definition 2.3.1 (Algebraic Interval [ÁDEK21, KN22])**.**
An *algebraic interval* is represented as a tuple $I = (I_l, I_u, I_L, I_U, I_{P_i}, I_\perp)$, where:

- I_l, I_u are algebraic numbers with $I_l \leq I_u$ that represent an interval over an $(i-1)$-dimensional sample point.
- I_L, I_U are sets of polynomials with the main variable x_i which vanish at $(s_1, \ldots, s_{i-1}, I_l)$ and $(s_1, \ldots, s_{i-1}, I_u)$ respectively.

- I_{P_i} is a set of polynomials with the main variable x_i which should be order-invariant in the constructed interval.
- $I_\perp$ is a set of lower-lever polynomials that must also be order-invariant in the underlying interval.

The bounds I_l, I_u of an algebraic interval are constant, but potentially algebraic, numbers. The sets of polynomials I_L, I_U define them in that they are multivariate polynomials which when evaluated at the sample point s became univariate with the bound as a real root.

Polynomials only matter so much as where they vanish. Based on this, we can define simplifications for sets of polynomials that stem from this fact.

Definition 2.3.2 (Standard CAD Simplifications[ÁDEK21])**.**

- Remove any constants or other polynomials that can easily be concluded never to equal zero.
- Normalize remaining polynomials, i.e., make the leading coefficient of each polynomial equal to 1. This ensures that we avoid storing multiple polynomials which define the same set of solutions over the real numbers.
- Store a square-free basis (a set of unique irreducible polynomials with distinct roots) for the factors rather than the polynomials themselves.

An NRA formula may also be denoted as a *Tarski Formula* [Bro99]. Meaning a Tarski formula is an integral polynomial constraint, i.e., a multivariate polynomial that is compared to zero with some comparison predicate, a Boolean constant `True`, `False`, a Boolean combination of Tarski formulas or a Tarski formula quantified via $\forall$ or $\exists$. An *Extended Tarski Formula (ETF)* is a Tarski formula with the addition of a new type of atomic formula that allows reference to the indexed roots of polynomials, which is defined in Definition 2.3.4.

Definition 2.3.3 (Indexed Root Expression [Bro99])**.**
Let $\alpha_1 < \ldots < \alpha_j$ be the distinct real roots of $p \in \mathbb{R}[x]$ in ascending order. We then define the $i-$th root of p as $\texttt{root}_i(p)$. If $i > j$ or $i < 1$, then $\texttt{root}_i(p)$ is undefined. For $p \in \mathbb{Z}[x_1, \ldots, x_n]$ and $i \in \mathbb{N}$ with $i \neq 0$, we define the *Indexed Root Expression (IRE)* $\texttt{root}(p,i,x_k)$ at the point $s = (s_1, \ldots, s_n) \in \mathbb{R}^n$ as the i-th distinct real root of $p(s_1, \ldots, s_{k-1}, x_k)$. The roots are ordered from smallest to largest. If the level of p is not k, if $p(s_1, \ldots, s_{k-1}, x_k)$ is the zero-polynomial or if it has fewer than i distinct real roots, then $\texttt{root}(p,i,x_k)$ is undefined.

Definition 2.3.4 (Extended Tarksi Formula (ETF) [Bro99])**.**
An ETF is a Tarski formula with the addition of atomic formulas of the form $q \bowtie \texttt{root}(p, i, x_k)$ where $p,q \in \mathbb{Z}[x_1, \ldots, x_n]$, $i \neq 0$, $0 < k \leq n$ and $\bowtie \in \{\leq, <, =, \neq, >, \geq\}$. At a point $s = (s_1, \ldots, s_n) \in \mathbb{R}^n$, the atomic formula $q \bowtie \texttt{root}(p,i,x_k)$ evaluates to `True` iff $\texttt{root}(p,i,x_k)$ with respect to s is defined and it holds that $q(s_1, \ldots, s_n) \bowtie \texttt{root}_i(p(s_1, \ldots, s_{k-1}, x, s_{j+1}, \ldots, s_n))$. Otherwise, it is `False`.

An ETF may be transformed into a Tarski formula, but it is, however, convenient to have the ability to refer to the roots of polynomials [Bro99]. In the main part of this thesis, we also need the concept of an *implicant*. An implicant ψ of a formula φ is a formula that simplifies φ while still implying it.

Definition 2.3.5 (Implicant [KN22])**.**
The formula ψ is an implicant of the formula φ iff $\psi \Rightarrow \varphi \wedge \texttt{constraints}(\psi) \subseteq \texttt{constraints}(\varphi)$. This concept can be extended to use a (partial) sample point $s \in \mathbb{R}^i$ as follows.

If $\varphi[s] = True$, then ψ is an implication of φ with respect to s iff:

$$\psi[s] = True \wedge (\psi \Rightarrow \varphi) \wedge \texttt{constraints}(\psi) \subseteq \texttt{constraints}_i(\varphi)$$

If $\varphi[s] = False$, then ψ is an implication of φ with respect to s iff:

$$\psi[s] = True \wedge (\psi \Rightarrow \neg\varphi) \wedge \texttt{constraints}(\psi) \subseteq \texttt{constraints}_i(\varphi)$$

We call ψ a *prime* implicant of φ if $\texttt{constraints}(\psi)$ is minimal among all implicants of φ.

An implicant allows for the creation of a more concise formula that may be simpler and more efficient to work with while preserving its logical meaning. Later, we are interested in computing the implicant of a formula $\bar{\varphi}$ with respect to a sample point $s \times s_i$. The implicant might include polynomials with the main variable x_i or lower, effectively constructing a characterization not only in the variable x_i but also for variables with a lower level. If $\bar{\varphi}[s \times s_i] = \texttt{False}$ and $\bar{\varphi}$ is a simple conjunction, then it is easy to compute the prime implicant of $\bar{\varphi}$ with respect to $s \times s$ as the negation of a single conflicting constraint in $\texttt{constraints}(\bar{\varphi})$. If $\bar{\varphi}[s \times s_i] = \texttt{True}$ and $\bar{\varphi}$ is a simple conjunction, then $\bar{\varphi}$ itself is its only prime implicant.

Chapter 3

Cylindrical Algebraic Covering for Quantifiers

In this chapter, we are presenting an algorithm that builds on the CAlC algorithm for the existential fragment of non-linear real arithmetic, as presented in Section 2.3. It extends its capabilities to solve the quantifier elimination problem and the decision problem for non-linear real arithmetic. All following algorithms and definitions are taken from [KN22], where the CAlC algorithm for quantifiers was first described. The algorithm adds the ability to process universally quantified variables. Handling existentially quantified variables remains the same in that a sample point is guessed. If the sample point cannot be extended to be a satisfying witness of the formula, it is generalized to an algebraic interval that is unsatisfiable for the same reasons. The next sample point is then picked outside of the collection of unsatisfiable cells. If the sample point can be extended to a satisfying witness, we can conclude satisfiability for the current dimension without guessing more points outside the collection of unsatisfiable intervals. If the collection of the unsatisfiable intervals covers the whole number line, and no new sample can be guessed, unsatisfiability can be concluded for this dimension. The idea of handling universally quantified variables is similar. A sample point is guessed, and if it can be extended to be a satisfying witness, it is generalized to an interval that is satisfiable for the same reasons. It is added to a collection of satisfiable intervals. Then, another sample point outside of the collected satisfiable cells is picked. If the chosen sample point cannot be extended to be a satisfying witness, the algorithm for a universally quantified variable can conclude the unsatisfiability of this dimension without the need to pick more sample points in this dimension. If the collection of satisfiable intervals covers the whole number line and no new sample can be guessed, satisfiability is concluded for the current dimension. The idea of handling parameters for the quantifier elimination problem is similar to how universally or existentially quantified variables are handled. The difference is that we collect both satisfiable and unsatisfiable intervals, and we do not stop processing a dimension if a sample point can or cannot be extended to be a satisfying witness. By doing this, we ensure that all satisfiable intervals are gathered. The satisfiable intervals are then described using an indexed root expression, as defined in Definition 2.3.3, and logically combined to form the respective result for the quantifier elimination problem. In the following, we present all algorithms and necessary sub-algorithms of the CAlC algorithm for quantifiers in detail.

3.1 Algorithms

This section focuses on the algorithms for the decision problem for formulas of nonlinear real arithmetic. Specifically, the algorithms for handling universally and existentially quantified variables and the required sub-algorithms are presented. In the following, we assume that the input formula φ is in prenex normal form, as defined in Definition 2.2.3. The prefix of the formula φ is denoted by $Q_1x_1, \ldots, Q_nx_n$, and the matrix of the formula φ is denoted by $\bar{\varphi}$. Both of these are available globally in all the following algorithms. Note that the order of the variables in the prefix gives the variable ordering.

Algorithm 1: `user_call()` [KN22, Algorithm 1]

Data: Global prefix $Q_1x_1, \ldots, Q_nx_n$ and matrix $\bar{\varphi}$
Output: Either `SAT` or `UNSAT`

```
1 (f, O) := recourse(())                    // Algorithm 2
2 return f
```

Algorithm 2: `recourse`(s) [KN22, Algorithm 2]

Data: Global prefix $Q_1x_1, \ldots, Q_nx_n$ and matrix $\bar{\varphi}$
Input: Sample point $s = (s_1, \ldots, s_{i-1}) \in \mathbb{R}^{i-1}$
Output: $(\texttt{SAT}, I)$ or $(\texttt{UNSAT}, I)$ where $s \times I$ can or can not be extended to be a satisfying model for $\bar{\varphi}$ for any $s_i \in I$. In any case, the algebraic information attached to I describes how s can be generalized.

```
1 if Q_i = ∃ then
2 |  return exists(s)                        // Algorithm 3
3 else
4 |  return forall(s)                        // Algorithm 4
```

Firstly, in Algorithm 1, we present the user interface to the recursive Algorithm 2 with an empty sample point and extract the main return value. This value represents the result of the decision problem. Algorithm 2 checks the current quantifier of the variable of interest and calls out to Algorithm 3 or Algorithm 4 accordingly. We use the global prefix $Q_1x_1, \ldots, Q_nx_n$ to determine the quantification. Meaning that given the passed sample point $s = (s_1, \ldots, s_{i-1}) \in \mathbb{R}^{i-1}$, the quantifier that determines which algorithm to call is Q_i.

Algorithm 3: `exists`(s) [KN22, Algorithm 3]

Data: Global prefix $Q_1x_1, \ldots, Q_nx_n$ and matrix $\bar{\varphi}$
Input: Sample point $s = (s_1, \ldots, s_{i-1}) \in \mathbb{R}^{i-1}$
Output: (`SAT`, I) or (`UNSAT`, I) where $s \times I$ can or can not be extended to be a satisfying model for $\bar{\varphi}$ for any $s_i \in I$. In any case, the algebraic information attached to I describes how s can be generalized.

```
1  I_unsat := ∅
2  while ⋃_{I∈I_unsat} I ≠ ℝ do
3  |  s_i := sample_outside(I_unsat)
4  |  if φ̄[s × s_i] = False then
5  |  |  (f,O) := (UNSAT, get_enclosing_interval(s,s_i))   // Algorithm 5
6  |  else if φ̄[s × s_i] = True then
7  |  |  (f,O) := (SAT, get_enclosing_interval(s,s_i))    // Algorithm 5
8  |  else
9  |  |  (f,O) := recourse(s × s_i)          // Algorithm 2, recursive call
10 |  if f = SAT then
11 |  |  R := characterize_interval(s, O)                 // Algorithm 6
12 |  |  I := interval_from_characterization((s_1,...,s_{i-2}), s_{i-1}, R)
   |  |   // Algorithm 8
13 |  |  return (SAT, I)
14 |  I_unsat := I_unsat ∪ {O}                                  // f = UNSAT
15 R := characterize_covering(s, I_unsat)                    // Algorithm 7
16 I := interval_from_characterization((s_1,...,s_{i-2}), s_{i-1}, R)
   // Algorithm 8
17 return (UNSAT, I)
```

Algorithm 3 is called with a partial sample point. It implicitly holds that $\bar{\varphi}[s] \neq$ `False`. This algorithm always returns a satisfiability-invariant interval in the dimension of the caller. This is a change compared to the Coverings Algorithms presented in [ÁDEK21], where the caller computes the characterization. In Line 1, we initialize a collection $\mathbb{I}_{unsat}$ of unsatisfiable intervals. The central part of this algorithm is a while-loop, which terminates when this collection of intervals forms a covering for the whole number line. In Line 3, a point s_i is selected, which lays outside of the partial covering formed by the collection $\mathbb{I}_{unsat}$. Selecting a sample point s_i is always possible; otherwise, the while-loop would have terminated.
Then, different actions are taken according to how the given formula φ evaluates under the extended sample point $s \times s_i$. Each of the following actions returns a tuple (f, O), where f is either `SAT` or `UNSAT` and O is an algebraic interval, as defined in Definition 2.3.1.

Line 6, $\varphi[s \times s_i] =$ `True`: The formula evaluates to `True` under the extended sample point $s \times s_i$, and thus the flag f is set to `SAT`. Algorithm 5 is called to create an algebraic interval around s_i over s for $\bar{\varphi}$ which is satisfiable for the same reasons.

Line 4, $\varphi[s \times s_i] =$ `False`: The formula evaluates to `False` under the extended sample point $s \times s_i$, and thus the flag f is set to `UNSAT`. Algorithm 5 is called to create an algebraic interval around s_i over s for $\bar{\varphi}$ which is unsatisfiable for

the same reasons.

Line 9, $\varphi[s \times s_i]$ is inconclusive: The formula does not evaluate to a truth value under the extended sample point $s \times s_i$. It holds that $i < n$, where n is the highest level of a polynomial appearing in $\bar{\varphi}$. Thus, the sample point $s \times s_i$ is not fully dimensional with respect to the formula $\bar{\varphi}$. In this case, Algorithm 2 is recursively called with the extended sample point $(s \times s_i)$ to check whether it can be further extended to be a satisfying witness. The return value of this call is then returned.

Based on how the flag f is set, the information stored in the algebraic interval O is handled differently.

$f = \texttt{SAT}$ (Lines 10–13): The extended sample point $s \times s_i$ is a satisfying witness for the formula $\bar{\varphi}$. Thus, the requirements for the existential quantifier are met, and the algorithm returns the tuple $(\texttt{SAT}, I)$, where I is the interval created from Algorithms 6 and 8.

$f = \texttt{UNSAT}$ (Line 14): The extended sample point $s \times s_i$ is not a satisfying witness for the formula $\bar{\varphi}$. Thus, the requirements for the existential quantifier are not met, and the interval O is added to the collection of unsatisfiable intervals $\mathbb{I}_{unsat}$.

This is repeated in the while loop until either a satisfying witness is found or the collection of unsatisfiable intervals $\mathbb{I}_{unsat}$ forms a covering for the whole number line. In the latter case, the algorithm can conclude unsatisfiability for the current dimension. That is because the collection of unsatisfiable intervals $\mathbb{I}_{unsat}$ forms a covering for the whole number line, and thus, no new sample point s_i outside of the collection of unsatisfiable intervals $\mathbb{I}_{unsat}$ can be guessed. Thus, the requirements for the existential quantifier cannot be met. Then in Line 15, Algorithm 7 is called to create a set of polynomials R which characterizes the unsatisfiability of the formula $\bar{\varphi}$ over s. This set of polynomials is then used to create an interval I in dimension $i - 1$, which is unsatisfiable for the same reasons as the collection of unsatisfiable intervals $\mathbb{I}_{unsat}$. The tuple $(\texttt{UNSAT}, I)$ is then returned.

Algorithm 4: `forall`(s) [KN22, Algorithm 4]

Data: Global prefix $Q_1x_1, \ldots, Q_nx_n$ and matrix $\bar{\varphi}$
Input: Sample point $s = (s_1, \ldots, s_{i-1}) \in \mathbb{R}^{i-1}$ such that $\bar{\varphi}[s] \neq$ `False`
Output: $(\texttt{SAT}, I)$ or $(\texttt{UNSAT}, I)$ where $s \times I$ can or can not be extended to be a satisfying model for $\bar{\varphi}$ for any $s_i \in I$. In any case, the algebraic information attached to I describes how s can be generalized.

1 $\mathbb{I}_{sat} := \emptyset$
2 **while** $\bigcup_{I \in \mathbb{I}_{sat}} I \neq \mathbb{R}$ **do**
3 $\quad s_i :=$ `sample_outside`$(\mathbb{I}_{sat})$
4 $\quad$ **if** $\bar{\varphi}[s \times s_i] =$ *`False`* **then**
5 $\quad\quad (f,O) := (\texttt{UNSAT}, \texttt{get\_enclosing\_interval}(s,s_i))$ `// Algorithm 5`
6 $\quad$ **else if** $\bar{\varphi}[s \times s_i] =$ *`True`* **then**
7 $\quad\quad (f,O) := (\texttt{SAT}, \texttt{get\_enclosing\_interval}(s,s_i))$ `// Algorithm 5`
8 $\quad$ **else**
9 $\quad\quad (f,O) := \texttt{recourse}(s \times s_i)$ `// Algorithm 2, recursive call`
10 $\quad$ **if** $f =$ *`UNSAT`* **then**
11 $\quad\quad R := \texttt{characterize\_interval}(s, O)$ `// Algorithm 6`
12 $\quad\quad I := \texttt{interval\_from\_characterization}((s_1, \ldots, s_{i-2}), s_{i-1}, R)$ `// Algorithm 8`
13 $\quad\quad$ **return** $(\textit{UNSAT}, I)$
14 $\quad \mathbb{I}_{sat} := \mathbb{I}_{sat} \cup \{O\}$ `// f = SAT`
15 $R := \texttt{characterize\_covering}(s, \mathbb{I}_{sat})$ `// Algorithm 7`
16 $I := \texttt{interval\_from\_characterization}((s_1, \ldots, s_{i-2}), s_{i-1}, R)$ `// Algorithm 8`
17 **return** $(\textit{SAT}, I)$

Algorithm 4 is fundamentally similar to `exists`(s) and has identical structure. In Algorithm 3, a collection of unsatisfiable intervals is stored, and we can determine satisfiability by checking whether the extended sample point $s \times s_i$ is a satisfying witness or the collection $\mathbb{I}_{unsat}$ forms a covering of the real number line. The idea of the Algorithm 4 is analogous. A collection of satisfiable intervals is stored. We can determine satisfiability by checking whether the extended sample point $s \times s_i$ is not a satisfying witness or whether the collection of satisfying intervals covers the whole number line. Firstly, in Line 1, we initialize a collection $\mathbb{I}_{sat}$ of satisfiable intervals. Just like in the `exists`(s), we enter a while loop that terminates when the collection of intervals forms a covering for the whole number line. Then, different actions are taken according to how the given formula φ evaluates under the extended sample point $s \times s_i$. This is done in the same way as in Algorithm 3, and we finally have a tuple (f, O), where f is either `SAT` or `UNSAT` and O is an algebraic interval. Based on how the flag f is set, the information stored in the algebraic interval O is handled differently.

$f =$ `UNSAT` (Lines 10–13): The extended sample point $s \times s_i$ can not be extended to be a satisfying witness for the formula $\bar{\varphi}$. Thus, the requirements for the universal quantifier are violated, and the algorithm returns the tuple $(\texttt{UNSAT}, I)$, where I is the interval created from Algorithms 6 and 8.

$f =$ `SAT` (Line 14): The extended sample point $s \times s_i$ is a satisfying witness for

the formula $\bar{\varphi}$. Thus, the requirements for the universal quantifier are met, and the interval O is added to the collection of satisfiable intervals $\mathbb{I}_{unsat}$.

The while-loop terminates when the collection of intervals forms a covering for the whole number line, and thus, no new sample point can be guessed. The algorithm can conclude satisfiability for the current dimension. With the same reasoning as in `exists`(s), we call Algorithms 7 and 8 to create an algebraic interval I in dimension $i-1$ which is satisfiable for the same reasons as the collection of satisfiable intervals $\mathbb{I}_{sat}$. The tuple $(\texttt{SAT}, I)$ is then returned. Both `exists`(s) and `forall`(s) compute the characterization of a covering and the characterization of an interval in the dimension of the callee. This calculation of a characterization introduces a technical problem for the very first dimension $i = 1$, as we refer in the arguments to the function Algorithm 8 to the point s_{i-1}, which does not exist in this case. To solve this problem, we assume that a special placeholder value is returned in this case instead of an interval. Now, we present some of the sub-algorithms used in detail in Algorithms 3 and 4.

Algorithm 5: `get_enclosing_interval`(s, s_i) [KN22, Algorithm 5]

Data: Global matrix $\bar{\varphi}$
Output: A satisfiability-invariant algebraic interval I around s_i over s
1 $P :=$ `implicant_polynomials`$(\bar{\varphi}, s \times s_i)$
2 Perform standard CAD simplifications to P `// Definition 2.3.2`
3 $I :=$ `interval_from_characterization`(s, s_i, P) `// Algorithm 8`
4 **return** I

Algorithm 5 is used to create an algebraic interval I around a sample point s_i in dimension i. This interval is satisfiability-invariant for the formula $\bar{\varphi}$ over s. In detail this means that if $\bar{\varphi}[s \times s_i] =$ `False` then for all other points $s_i' \in I$ it holds that $\bar{\varphi}[s \times s_i'] =$ `False`. Analogously, if $\bar{\varphi}[s \times s_i] =$ `True` then for all other points $s_i' \in I$ it holds that $\bar{\varphi}[s \times s_i'] =$ `True`. A core part of Algorithm 5 is Line 1, in which the polynomials which are responsible for the truth value of the formula $\bar{\varphi}$ over $s \times s_i$ are extracted. This is done by computing the implicant, see Definition 2.3.5, of the formula $\bar{\varphi}$ over $s \times s_i$. We assume that the algorithm `implicant_polynomials` computes the (preferably prime) implicant ψ of the formula $\bar{\varphi}$ over $s \times s_i$ and returns all polynomials that occur in ψ.

Algorithm 6: `characterize_interval`(s, I) [KN22, Algorithm 6]

Input: Sample point $s \in \mathbb{R}^i$ and single algebraic interval I over s in dimension $i+1$
Output: Polynomials $R \subseteq \mathbb{Q}[x_1, \ldots, x_i]$ that characterize a satisfiability-invariant region around s

1 Extract: $l = I_l, u = I_u, L = I_L, U = I_U, P_{i+1} = I_{P_{i+1}}, P_\perp = I_{P_\perp}$
2 $R := P_\perp \cup \texttt{disc}(P_{i+1})$
3 $R := R \cup \{\texttt{required\_coefficients}(p) | p \in P_{i+1}\}$
`// [ÁDEK21, Algorithm 6]`
4 $R := R \cup \{\texttt{res}(p,q) | p \in L, q \in P_{i+1}, q(s \times \alpha) = 0 \text{ for some } \alpha \leq l\}$
5 $R := R \cup \{\texttt{res}(p,q) | p \in U, q \in P_{i+1}, q(s \times \alpha) = 0 \text{ for some } \alpha \geq l\}$
6 Perform standard CAD simplifications to R `// Definition 2.3.2`
7 **return** R

Algorithm 7: `characterize_covering`$(s, \mathbb{I})$ [KN22, Algorithm 7]

Input: Sample point $s \in \mathbb{R}^i$ and a covering of algebraic intervals $\mathbb{I}$ over s in dimension $i+1$.
Output: Polynomials $R \subseteq \mathbb{Q}[x_1, \ldots, x_i]$ characterizing a satisfiability-invariant region around s.

1 $\mathbb{I} := \texttt{compute\_cover}(\mathbb{I})$
2 $R := \bigcup_{I \in \mathbb{I}} \texttt{characterize\_interval}(s, I)$ `// Algorithm 6`
3 **for** $j \in \{1, \ldots, |\mathbb{I}| - 1\}$ **do**
4 $\quad R := R \cup \{\texttt{res}(p, q) | p \in U_j, q \in L_{j+1}\}$
5 Perform standard CAD simplifications to R `// Definition 2.3.2`
6 **return** R

Algorithm 6 is used to create a set of polynomials R which characterizes a satisfiability-invariant region in dimension i around a sample point s for a given algebraic interval I in dimension $i+1$. Similarly, Algorithm 7 is used to create a set of polynomials R which characterizes a satisfiability-invariant region in dimension i around a sample point s for a given covering of algebraic intervals $\mathbb{I}$ in dimension $i+1$. The sets of polynomials created by that algorithm are pruned and simplified as defined in Definition 2.3.2. In Algorithm 7 and line 1, we refer to a sub-algorithm `compute_cover` which is used to compute a covering based on the set of algebraic intervals $\mathbb{I}$.

Algorithm 8: `interval_from_characterization`(s, s_i, P) [ÁDEK21, Algorithm 5]

Input: Sample point $s \in \mathbb{R}^{i-1}$, an extension s_i to the sample, and a set of polynomials P in $\mathbb{Q}[x_1, \ldots, x_i]$ that characterize why $s \times s_i$ can or cannot be extended to be a satisfying witness.

Output: An Interval I around s_i such that on $s \times I$ the constraints are equisatisfiable for the same reasons as on $s \times s_i$.

1 $P_\perp := \{p \in R | p \in \mathbb{Q}[x_1, \ldots, x_{i-1}]\}$
2 $P_i := R \setminus P_\perp$
3 $Z := \{-\infty\} \cup$ `real_roots_with_check`$(P_i, s) \cup \{\infty\}$
4 $l := \max\{z \in Z | z \leq s_i\}$
5 $u := \min\{z \in Z | z \geq s_i\}$
6 $L := \{p \in P_i | p(s \times l) = 0\}$
7 $U := \{p \in P_i | p(s \times u) = 0\}$
8 Define new Interval I with $I_l = l, I_u = u, I_L = L, I_U = U, I_{P_i} = P_i, I_{P_\perp} = P_\perp$
9 **return** I

In Algorithm 8, we present how a characterization, given as a set of polynomials P, is used to expand the sample point to an interval. This interval is equisatisfiable to the sample point for the same reasons as the sample point. Note that `interval_from_characterization`(s, s_i, P) is called the same way for characterizations created by a single algebraic interval, in Algorithm 6, and for characterizations created by a covering of algebraic intervals, in Algorithm 7. For more details about the presented sub-algorithms, we refer to [ÁDEK21].

3.2 Quantifier Elimination

In the following section, we present how the coverings method can be used for the quantifier-elimination problem for non-linear real arithmetic, as defined in Definition 2.2.6. Given a formula φ in real arithmetic, we want to compute a quantifier-free formula ψ, which is equivalent to φ. In the following, we assume that the formula φ is given in prenex normal form, as defined in Definition 2.2.3. As presented in Chapter 2, we can interpret the free variables as parameters for the resulting formula when dealing with quantifier elimination. This means that the resulting formula is a quantifier-free formula in the parameters. These parameters are the only variables allowed to occur in the resulting formula. It is known that quantifier-elimination for non-linear real arithmetic is decidable, and the lower bound for the complexity of this problem is doubly exponential in the number of quantifier alternations [Tar51, DH88].

The main idea of the following algorithm is to consider the parameters first and treat them similarly to existentially or universally quantified variables, with a few differences. Instead of returning early, when a requirement for the quantifier is met or violated, we collect both satisfiable and unsatisfiable intervals until the whole number line is covered. This ensures that all satisfiable intervals in the parameter space are constructed. Each interval is translated into a formula, built of constraints over indexed-root expressions, as defined in Definition 2.3.3. The constructed formula describes exactly the satisfying algebraic intervals in the parameter space.

Algorithm 9: `parameter`(s) [KN22, Algorithm 9]

Data: Global prefix $Q_{k+1}x_{k+1}, \ldots, Q_n x_n$ and matrix $\bar{\varphi}$
Input: Sample point $s = (s_1, \ldots, s_{i-1}) \in \mathbb{R}^{i-1}$
Output: (ψ, I) where ψ characterizes all satisfying intervals over s within $s \times I$

```
1  𝕀 := ∅
2  ψ := False
3  while ⋃_{I∈𝕀_sat} I ≠ ℝ do
4  |  s_i := sample_outside(𝕀_sat) if φ̄[s × s_i] = False then
5  |  |  (T,O) := (False, get_enclosing_interval(s,s_i)) // Algorithm 5
6  |  else if φ̄[s × s_i] = True then
7  |  |  (T,O) := (True, get_enclosing_interval(s,s_i))  // Algorithm 5
8  |  else if i < k then
9  |  |  (T,O) := parameter(s × s_i)        // Algorithm 9, recursive call
10 |  else // It holds that k ≤ i < n
11 |  |  (f,O) := recourse(s × s_i)          // Algorithm 2, recursive call
12 |  |  if f = SAT then
13 |  |  |  T := True
14 |  |  else
15 |  |  |  T := False
16 |  𝕀 := 𝕀 ∪ {O}
17 |  ψ := ψ ∨ (indexed_root_formula(O, s) ∧ T)
18 R := characterize_covering(s, 𝕀_sat)                    // Algorithm 7
19 I := interval_from_characterization((s_1, ..., s_{i-2}), s_{i-1}, R)
     // Algorithm 8
20 return (ψ, I)
```

Given a formula $\varphi := Q_{k+1}x_{k+1}, \ldots, Q_n x_n.\bar{\varphi}$ in prenex normal form with $k \neq 0$, Algorithm 9 is called with an empty sample point. The return value of this initial call is a tuple (ψ, I), where ψ is a quantifier-free formula in the parameters of φ, equivalent to φ. Algorithm 9 is similar to Algorithm 3 and Algorithm 4, but with a few differences. In Lines 1 and 2, a collection of algebraic intervals in the given dimension and the respective output formula of the given dimension is initialized. The collection of algebraic intervals $\mathbb{I}$ is used to collect all satisfiable and unsatisfiable cells in the given dimension. This ensures that all satisfiable regions of the parameter space are enumerated. The formula ψ describes all satisfiable regions in the given dimension. The main part of the algorithm, a while-loop, terminates when the collection of algebraic intervals $\mathbb{I}$ forms a covering for the whole number line. In Line 4, a point s_i is selected, outside the partial covering formed by the collection $\mathbb{I}$. Afterward, similar to Algorithms 3 and 4, the formula $\bar{\varphi}$ is evaluated under the extended sample point $s \times s_i$. Different actions are taken depending on the result of this evaluation and the current dimension i. Each of the following actions defines a tuple (T, O), where T is the formula in the parameters and O is an algebraic interval.

> $\bar{\varphi}[s \times s_i] =$ `True`(*Line* 4); $\bar{\varphi}[s \times s_i] =$ `False`(*Line* 6): The formula $\bar{\varphi}$ evaluates to a truth value under the extended sample point $s \times s_i$, and thus the flag T is set to respectively `True` or `False`. Algorithm 5 is called to create an algebraic interval around s_i over s for $\bar{\varphi}$ which is satisfiable or unsatisfiable for the same

reasons.

$i < k$(*Line* 8): It holds that $\bar{\varphi}[s \times s_i]$ is inconclusive and $i < k$ meaning that the x_{i+1} is free. In this case, Algorithm 9 is called with the extended sample point $(s \times s_i)$. The return value of this recursive call is exactly the tuple (T, O).

$k \leq i < n$(*Line* 11): It holds that $\bar{\varphi}[s \times s_i]$ is inconclusive and $k \leq i < n$. This means that the variable x_{i+1} is not a parameter but instead either universally or existentially quantified. In this case, Algorithm 2 is called with the extended sample point $(s \times s_i)$. The return value of this call is a tuple (f, O), where f is either `SAT` or `UNSAT` and O is an algebraic interval. Depending on the value of f, the flag T is set to `True` or `False`.

As $\mathbb{I}$ is a collection of both satisfiable and unsatisfiable cells, we add the algebraic interval O to the collection $\mathbb{I}$ in any case. We add the newly gained information of the algebraic interval O and the formula T to the formula ψ. To do this, we call Algorithm 10 with the algebraic interval O, which outputs a formula describing the bounds of the algebraic interval. This formula is then added to the formula T with a conjunction. This conjunction is then added to the formula ψ with a disjunction.

The while-loop terminates when the collection of intervals forms a covering for the whole number line. Equivalently to Algorithms 3 and 4, the collection of algebraic intervals $\mathbb{I}$ used to create an algebraic interval in dimension $i-1$ by calling Algorithms 7 and 8. This interval is then returned as the second element of the tuple (ψ, I).

Algorithm 10: `indexed_root_formula`(O) [KN22, Chapter 4]

Input: Algebraic interval O, Sample point $s = (s_1, \ldots, s_{i-1}) \in \mathbb{R}^{i-1}$
Output: Formula ψ describing the bounds of the algebraic interval O symbolically using constraints over indexed-root expressions

1 $\psi := \texttt{True}$
2 **foreach** $p \in I_L$ **do**
3 | Choose $j_{p,l}$ such that $\text{root}[p, j_{p,l}](s_1, \ldots, s_{i-1}) = I_l$
4 | $\psi := \psi \wedge \text{root}[p, j_{p,l}] < x_i$
5 **foreach** $p \in I_U$ **do**
6 | Choose $j_{p,u}$ such that $\text{root}[p, j_{p,u}](s_1, \ldots, s_{i-1}) = I_u$
7 | $\psi := \psi \wedge x_i < \text{root}[p, j_{p,u}] < x_i$
8 **return** ψ

Algorithm 10 is used to translate an algebraic interval O into its respective formula over indexed-root expressions (Definition 2.3.3). To do this, we use constraints over indexed-root expressions to describe the bounds of the algebraic interval O symbolically, as defined in Definition 2.3.3. These indexed-root expressions represent the polynomial sets that describe the bounds of a satisfiable algebraic interval. When combined using Boolean operators, the resulting formula is an ETF, as defined in Definition 2.3.4.

Chapter 4

Handling Constraints Independently

In the following chapter, we will present an extension to the cylindrical algebraic coverings algorithm for quantifiers. This approach is strictly based on the algorithms presented in Chapter 3. Likewise, as introduced in Chapter 3, we have an input formula φ in prenex normal form. This results in the fact that when we need to evaluate the formula φ at a sample point s, we continually evaluate the whole formula φ at s. Similarly, when computing the implicant, we must compute the implicant of the whole formula φ. Under certain circumstances, however, it is possible to *split* the formula φ into two or more independent subformulas, which can be processed separately. To illustrate this, consider the following example: Assume that we have a formula $\varphi := Q_1x_1.Q_2x_2.Q_3x_3.Q_4x_4.(\varphi_1(x_1,x_4)\ \sigma\ \varphi_2(x_2,x_3))$ with $\sigma \in \{\vee, \wedge\}$ and $Q_i \in \{\exists, \forall\}$ in prenex normal form and the variable ordering $x_1 \prec x_2 \prec x_3 \prec x_4$. Thus, it holds that the subformulas φ_1 and φ_2 are quantifier-free. Firstly, observe that φ_1 and φ_2 do not share any variables. This has the effect that any assignment to x_1 or x_4 will not affect the result when solving φ_2. Analogously, any assignment to x_2 or x_3 will not affect the result when solving φ_1. Thus, without any loss of information, we can split the formula φ and process the subformulas φ_1 and φ_2 independently. This has the effect that we can process the subformulas φ_1 and φ_2 separately using the cylindrical algebraic coverings algorithm presented in Chapter 3. If we assume that $\sigma = \vee$ and we can already determine the satisfiability of φ_1, we can also determine the satisfiability of the whole formula φ without having to process the subformula φ_2. Likewise, suppose we assume that $\sigma = \wedge$ and we can already determine the unsatisfiability of φ_1. In that case, we can also determine the unsatisfiability of the whole formula φ without having to process the subformula φ_2. In other cases, however, we must also process φ_2. The results of processing φ_1 and φ_2 can then be combined. To show a possible advantage of this approach, consider that the subformula φ_2 is highly complex and may require a lot of computational resources to be processed. Then, if we can determine the satisfiability of φ_1 without processing φ_2, we can save a lot of computational resources. This is the main idea behind the approach presented in this chapter. In the example we presented above, the subformulas φ_1 and φ_2 do not share any variables; however, this is not necessary, as we show later. It suffices that the subformulas share at most one variable, the respectively next variable in the variable ordering given a partial sample point s. In the following, we formally define

what it means for two formulas to be independent with respect to a variable x_i and a (partial) sample point s.

Definition 4.0.1 (Independent Formulas)**.**
We call two formulas φ, φ' independent with respect to a variable x_i and a (partial) sample point $s = (s_1, \ldots, s_{i-1}) \in \mathbb{R}^{i-1}$ if they do not share any variables except x_i when evaluated at s. Formally, that is φ *and* φ' *are independent with respect to* x_i *and* s iff $(\texttt{vars}(\varphi[s]) \cap \texttt{vars}(\varphi'[s])) \setminus \{x_i\} = \emptyset$.

In Example 4.0.1, we show examples of independent formulas.

Example 4.0.1. Independent Formulas
Let $\varphi_1(x_1,x_2,x_3) := x_1 \cdot x_3 + x_2 > 0$ and $\varphi_2(x_1,x_2,x_3) := x_1 + x_2 + x_3 > 0$.

$s := ()$: We want to check if φ_1, φ_2 are independent with respect to variable x_1 and s. It holds that $\varphi_1[s] = x_1 \cdot x_3 + x_2 > 0$ and $\varphi_2[s] = x_1 + x_2 + x_3 > 0$. Thus $(\texttt{vars}(\varphi_1[s]) \cap \texttt{vars}(\varphi_2[s])) \setminus \{x_1\} = (\{x_1, x_2, x_3\} \cap \{x_1,x_2,x_3\}) \setminus \{x_1\} = \{x_2, x_3\} \neq \emptyset$ Thus, φ_1 and φ_2 are not independent with respect to x_1 and s.

$s := (x_1 = 0)$: We want to check if φ_1, φ_2 are independent with respect to variable x_2 and s. It holds that $\varphi_1[s] = x_2 > 0$ and $\varphi_2[s] = x_2 + x_3 > 0$. Thus $(\texttt{vars}(\varphi_1[s]) \cap \texttt{vars}(\varphi_2[s])) \setminus \{x_2\} = (\{x_2\} \cap \{x_2,x_3\}) \setminus \{x_2\} = \emptyset$ Thus, φ_1 and φ_2 are independent with respect to x_2 and s.

Now that we have introduced the notion of independent formulas, we can formally define what it means to *split* a formula. That is, we split the given formula into two or more independent subformulas that are pairwise independent with respect to a variable x_i and a (partial) sample point s.

Definition 4.0.2 (Formula Split)**.**
Given a set of formulas $\Phi := \{\varphi_1, \ldots, \varphi_n\}$, a variable x_i, a sample $s = (s_1, \ldots, s_{i-1}) \in \mathbb{R}^{i-1}$ we can partition the sets as follows: Let $\texttt{split}(\Phi, x_i, s)$ be the coarsest partitioning $\{A_1, \ldots, A_j\}$ of Φ such that for all $i \neq j$ and $\varphi \in A_i, \varphi' \in A_j$ it holds that φ and φ' are independent with respect to x_i and s. We can combine the sets in $\texttt{split}(\Phi, x_i, s)$ using a Boolean connective $\sigma \in \{\vee, \wedge\}$ as follows: $\texttt{split\_formulas}(\Phi, x_i, s, \sigma) := \{\sigma_{\varphi \in \Phi_i}(\varphi) | \Phi_i \in \text{split}(\Phi, x_i, s)\}$

In Definition 4.0.2, we have formally defined how to split a set of formulas Φ with respect to a variable x_i and a (partial) sample point s. Now, let φ be a formula in prenex normal form with the matrix $\bar{\varphi} = \varphi_1 \sigma \ldots \sigma \varphi_n$. Then, we denote $\texttt{split\_formulas}(\{\varphi_1, \ldots, \varphi_n\}, x_i, s, \sigma)$ as the formula split of φ with respect to x_i and s. Now, we need to define another precondition for a formula split to be *viable*. That is the quantification of the variable x_i in combination with the used Boolean connective σ. Only 2 combinations of variable quantification and Boolean connective are possible. This is shown in Definition 4.0.3.

Definition 4.0.3 (Viable Quantification and Boolean Connective)**.**
Given a set of formulas $\Phi := \{\varphi_1, \ldots, \varphi_n\}$, a variable x_i, its respective quantifier $Q_i \in \{\forall, \exists\}$, a sample $s = (s_1, \ldots, s_{i-1}) \in \mathbb{R}^{i-1}$ and a Boolean connective $\sigma \in \{\vee, \wedge\}$. Then, a formula split is only possible for the following combinations of quantifier and Boolean connective:

- $Q_i = \exists$ and $\sigma = \vee$

- $Q_i = \forall$ and $\sigma = \wedge$

In the following, we will say that a split is *viable* if the combination of quantifier and Boolean connectives are given according to Definition 4.0.3. If a split is not viable, we set $\texttt{split}(\Phi, x_i, s, \sigma) := \{\Phi\}$.

Example 4.0.2 (Formula Split)**.**
Let $\varphi := \forall x_1.\exists x_2.\forall x_3.\underbrace{x_1 \cdot x_3 + x_2 > 0}_{:=\varphi_1(x_1,x_2,x_3)} \vee \underbrace{x_1 + x_2 + x_3 > 0}_{:=\varphi_2(x_1,x_2,x_3)}$. The formula φ is in prenex normal form, and we have $\sigma = \vee$. Note that φ_1,φ_2 are also used in Example 4.0.1.

$s = ()$: According to Definition 4.0.3 this split is not viable, as we have $Q_1 = \forall$ and $\sigma = \vee$. Thus we have $\texttt{split}(\{\varphi_1, \varphi_2\}, x_1, ()) = \{\{\varphi_1, \varphi_2\}\}$ and $\texttt{split\_formulas}(\{\varphi_1, \varphi_2\}, x_1, (), \vee) = \{\varphi_1 \vee \varphi_2\}$.

$s = (x_1 = 0)$: According to Definition 4.0.3 this split is viable, as we have $Q_1 = \exists$ and $\sigma = \vee$. We have $\texttt{split}(\{\varphi_1, \varphi_2\}, x_1, (x_1 = 0)) = \{\{\varphi_1\}, \{\varphi_2\}\}$ and $\texttt{split\_formulas}(\{\varphi_1, \varphi_2\}, x_1, (x_1 = 0), \vee) = \{\varphi_1, \varphi_2\}$.

Theorem 4.0.1.
Let φ be a formula in prenex normal form with $\bar{\varphi} = \varphi_1 \sigma \dots \sigma \varphi_n$, with $\sigma \in \{\vee, \wedge\}$ and thus with the subformulas $\Phi := \{\varphi_1, \dots, \varphi_n\}$. Further, let $x_i \in \texttt{vars}(\varphi)$ and let $s = (s_1, \dots s_{i-1}) \in \mathbb{R}^{i-1}$ be a partial sample point such that a split is viable, according to Definition 4.0.3. Let $\texttt{split\_formulas}(\Phi, x_i, s, \sigma) = \{\varphi'_1, \dots, \varphi'_m\}$ be the formula split of Φ with respect to x_i and s. Then the following holds:

$Q_i = \exists$: If and only if there exists a formula $\varphi'_j \in \{\varphi'_1, \dots, \varphi'_m\}$ such that φ'_j is satisfiable, then φ is also satisfiable.

$Q_i = \forall$: If and only if there exists a formulas $\varphi'_j \in \{\varphi'_1, \dots, \varphi'_m\}$ such that φ'_j is unsatisfiable, then φ is also unsatisfiable.

Proof. To prove this theorem, we consider the cases separately.

- Let x_i be existentially quantified. If there exists a formula $\varphi' \in \texttt{split\_formulas}(\Phi, x_i, s, \sigma)$ such that φ' is satisfiable, then there exists an extension s' to the sample point s such that $\varphi'[s'] = \texttt{True}$. Because the split is viable it must hold that the original function φ is a disjunction. It thus holds that $\varphi[s'] = \texttt{True}$ and is thus also satisfiable. The other direction is analogous. If φ is satisfiable, then an extension to the sample point s, namely s', exists, such that $\varphi[s'] = \texttt{True}$. As φ is a disjunction, there must exist a formula $\varphi' \in \{\varphi'_1, \dots, \varphi'_m\}$ such that $\varphi'[s'] = \texttt{True}$. Thus, the reason why φ is satisfiable is the same as why φ' is satisfiable.

- Let x_i be universally quantified. If there exists a formula $\varphi' \in \texttt{split\_formulas}(\Phi, x_i, s, \sigma)$ such that φ' is unsatisfiable, then there does does not exist an extension s' to the sample point s such that $\varphi'[s'] = \texttt{True}$. Thus, for every fully dimensional extension s' of s it must holds that $\varphi'[s'] = \texttt{False}$. Because the split is viable it must hold that the original function φ is a conjunction. It thus holds that $\varphi[s'] = \texttt{False}$ for all fully dimensional extension s' of s and thus φ is also unsatisfiable. The other direction is analogous. If φ is unsatisfiable, there does not exist an extension s' of the sample point s such that $\varphi[s'] = \texttt{True}$. Thus for every fully dimensional extension s' of s it must hold that $\varphi[s'] = \texttt{False}$.

As φ is a conjunction, it must hold that there must exist $\varphi' \in \{\varphi'_1, \ldots, \varphi'_m\}$ for which it holds that $\varphi'[s'] = \texttt{False}$. Thus, the reason why φ is unsatisfiable is the same as why φ' is unsatisfiable.

□

To see why any other combination of quantifier and σ are not viable, consider the examples shown in Example 4.0.3.

Example 4.0.3 (Bad Formula Splits)**.**

- Given a formula $\varphi := \exists x.x = 0 \land x = 1$. Thus, we have the set of formulas $\Phi = \{x = 0, x = 1\}$, the variable x, the sample point $s = ()$ and the Boolean connective $\sigma = \land$. According to Definition 4.0.2, we can split the set of formulas Φ with respect to x and s as follows: $\texttt{split\_formulas}(\Phi, x, s) = \{\{x = 0\}, \{x = 1\}\}$. This implies that we can solve $\exists x.x = 0$ and $\exists x.x = 1$ independently and combine the results using the Boolean connective $\land$. As both $\exists x.x = 0$ and $\exists x.x = 1$ are satisfiable, we would conclude that $\exists x.x = 0 \land x = 1$ is satisfiable, which is wrong.

- Given a formula $\varphi := \forall x.x < 1 \lor x > -1$. Thus, we have the set of formula $\Phi = \{x < 1, x > -1\}$, the variable x, the sample point $s = ()$ and the Boolean connective $\sigma = \lor$. According to Definition 4.0.2, we can split the set of formulas Φ with respect to x_1 and s as follows: $\texttt{split\_formulas}(\Phi, x, s) = \{\{x < 1\}, \{x > -1\}\}$. This implies that we can solve $\forall x.x < 1$ and $\forall x.x > -1$ independently and combine the results using the Boolean connective $\lor$. As both $\forall x.x < 1$ and $\forall x.x > -1$ are unsatisfiable, we would conclude that $\forall x.x < 1 \lor x > -1$ is unsatisfiable, which is wrong.

In the following, we are going to present Algorithm 11, which, given a set of formulas Φ, a variable x_i, a sample point $s \in \mathbb{R}^{i-1}$ returns exactly the result $\texttt{split}(\Phi, x_i, s)$ according to Definition 4.0.2. To do this, we will transform the problem into a graph problem and then use a graph algorithm to find the connected components of the given graph. A *connected component* of a given undirected graph is defined as a connected subgraph that is not part of any larger connected subgraph. Every vertex $v \in V$ of a graph belongs to one and only one component, which may be found as the induced subgraph of the set of vertices reachable from v [JA95].

Algorithm 11: $\texttt{split}(\Phi, x_i, s)$

Input: A set of formulas Φ, a variable x_i and a sample point $s \in \mathbb{R}^{i-1}$
Output: A set of sets of formulas representing exactly the formula split of Φ with respect to x_i and s

```
1 G := (V, E)                                          // Initialize Graph
2 V := Φ                                   // Vertices are exactly the formulas
3 foreach φ_i ∈ Φ do
4 |   for φ_j ∈ Φ such that j > i do
5 |   |   if φ_i is not independent of φ_j with respect to x_i and s then
6 |   |   |   E := E ∪ {(φ_i, φ_j)}
7 Compute vertices C of the connected compontents of G
8 return C
```

Given a set of formulas Φ, a variable x_i and a sample $s = (s_1, \ldots, s_{i-1}) \in \mathbb{R}^{i-1}$, the graph G we are going to use is defined as follows:

- The vertices of the graph are exactly the formulas in Φ.
- Edge between two vertices φ, φ' iff φ and φ' are not independent with respect to x_i and s.

The graph G is constructed in Lines 1–6. Then, in Line 7, the connected components of G are computed. This can be done, for example, using a DFS-based approach with a time complexity of $\mathcal{O}(|V| + |E|)$ [JA95, Sch11]. The subgraphs resulting from this computation are of interest only for the set of vertices they contain. We call the set of vertices of the connected components C. Now, to show that C is exactly $\texttt{split}(\Phi, x_i, s)$, we need to show that for $C_i, C_j \in C$ with $i \neq j$ it holds that $\varphi \in C_i$ and $\varphi' \in C_j$ are independent with respect to x_i and s. To see this, let $\varphi \in C_i$ and $\varphi' \in C_j$ with $i \neq j$ be formulas from two different connected components. Then, by definition of a connected component, no path exists from φ to φ' in G. Thus, there is no edge from φ to φ' in G. By construction of G, this means that φ and φ' are independent with respect to x_i and s. Thus, the vertices C of the connected components of G are exactly $\texttt{split}(\Phi, x_i, s)$. This set of vertices C is then returned in Line 8.

Example 4.0.4 (Formula Split)**.**
Let $\varphi := x_1 \cdot x_2 - x_4 + 3 > 0 \vee x_2 - x_3 > 0 \wedge x_2^2 \neq 0 \vee 4 \cdot x_3^2 + x_3 \cdot x_2 - 7 > 0$ and we have the variable ordering $x_1 \prec x_2 \prec x_3 \prec x_4$ and the sample point $s = ()$. Then we can set:

$$\varphi_1(x_1, x_2, x_4) := x_1 \cdot x_2 - x_4 + 3 > 0$$

$$\varphi_2(x_2, x_3) := x_2 - x_3 > 0 \wedge x_2^2 \neq 0$$

$$\varphi_3(x_2, x_3) := 4 \cdot x_3^2 + x_3 \cdot x_2 - 7 > 0$$

Thus, we have $\Phi = \{\varphi_1, \varphi_2, \varphi_3\}$, as $s = ()$ and x_1 is the variable of interest. Thus, we are interested in computing the formula split of Φ with respect to x_1 and s, or $\texttt{split\_formulas}(\Phi, x_1, (), \vee)$. Then we can construct the graph $G_1 = (V,E)$ according to Algorithm 11 as follows:

1. We set $V := \Phi = \{\varphi_1, \varphi_2, \varphi_3\}$.
2. Iterate over each pair of formulas $\varphi_i, \varphi_j \in \Phi$ with $i < j$ and check if the formulas are independent with respect to x_1 and s. As $s = ()$, the formulas are independent with respect to x_1 and s iff they do not share any variables except x_1.
   - $\varphi_1(x_1, x_2, x_4)$ and $\varphi_2(x_2, x_3)$ are not independent with respect to x_1 and s as they share the variable $\{x_2\}$.
   - $\varphi_1(x_1, x_2, x_4)$ and $\varphi_3(x_2, x_3)$ are not independent with respect to x_1 and s as they share the variable $\{x_2\}$.
   - $\varphi_2(x_2, x_3)$ and $\varphi_3(x_2, x_3)$ are not independent with respect to x_1 and s as they share the variables $\{x_2, x_3\}$.

   Thus, we have $E = \{(\varphi_1, \varphi_2), (\varphi_1, \varphi_3), (\varphi_2, \varphi_3)\}$. The resulting graph $G_1 = (V, E)$ is shown in Figure 4.1a.

(a) Graph G_1 (b) Graph G_2

Figure 4.1: Graphs G_1 and G_2 resulting from Example 4.0.4

3. Compute the connected components of G. As G is a complete graph, there is only one connected component, which is $C = \{\varphi_1, \varphi_2, \varphi_3\}$.

4. Combine the formulas in C using the Boolean connective $\vee$. This results in the formula $\varphi_1 \vee \varphi_2 \vee \varphi_3$.

In total, the formula could not be split into two or more independent subformulas. Now, we consider the same formula φ and the same variable ordering $x_1 \prec x_2 \prec x_3 \prec x_4$, but a different sample point $s = (x_1 = 1)$. Thus, we are now interested in computing the formula split of φ with respect to x_2 and s, or `split_formulas`$(\Phi, x_2, (x_1 = 1), \vee)$. Then we can construct the graph $G_2 = (V,E)$ according to Algorithm 11 as follows:

1. Again we set $V := \Phi = \{\varphi_1, \varphi_2, \varphi_3\}$.

2. Iterate over each pair of formulas $\varphi_i, \varphi_j \in \Phi$ with $i < j$ and check if the formulas are independent with respect to x_1 and s. As $s = (x_1 = 1)$, the formulas are independent with respect to x_1 and s iff they do not share any variables except x_2 when evaluated at s.

   - $\varphi_1(1, x_2, x_4)$ and $\varphi_2(x_2, x_3)$ are independent with respect to x_2 and s as they only share the variable $\{x_2\}$.
   - $\varphi_1(1, x_2, x_4)$ and $\varphi_3(x_2, x_3)$ are independent with respect to x_2 and s as they only share the variable $\{x_2\}$.
   - $\varphi_2(x_2, x_3)$ and $\varphi_3(x_2, x_3)$ are not independent with respect to x_2 and s as they share the variables $\{x_2, x_3\}$.

   Thus, we have $E = \{(\varphi_2, \varphi_3)\}$. The resulting graph $G_2 = (V, E)$ is shown in Figure 4.1b.

3. Compute the connected components of G. Vertex φ_1 is not connected to any other vertices in G and thus forms a single connected component. Vertices φ_2 and φ_3 are connected and thus form a connected component. Thus, we have $C = \{\{\varphi_1\}, \{\varphi_2, \varphi_3\}\}$.

4. Combine the formulas in C using the Boolean connective $\vee$. This results in the formulas $\{\varphi_1, \varphi_2 \vee \varphi_3\}$.

In total, the formula could be split into two independent subformulas, namely φ_1 and $\varphi_2 \vee \varphi_3$. Note that the actual formulas $\varphi_1, \varphi_2, \varphi_3$ are not crucial for the computation of the formula split, only the variables they contain.

4.1 Adapting the Cylindrical Algebraic Coverings Algorithm

In the following, we will show how the cylindrical algebraic coverings method algorithm, as presented in Chapter 3, can be adapted to handle the concept of formula splits. The idea of the adaptions is to split the formula φ with respect to a variable x_i and a partial sample point s into two or more independent subformulas whenever possible. These independent subformulas are then processed further using the cylindrical algebraic coverings algorithm. When the result of a formula from this split determines the satisfiability of φ for the variable x_i, we can return the result without processing the other formulas from the split. If not, and we have processed all independent subformulas, we must combine the individual results. Intuitively, how to handle the individual results of the independent subformulas depends on the Boolean connective σ used to combine the subformulas. This is shown in Definition 4.1.1.

Definition 4.1.1 (Combining two independent formulas)**.**
Let φ be a formula in prenex normal form, with $\bar{\varphi} := \varphi_1 \sigma \varphi_2$ and $\sigma \in \{\vee, \wedge\}$. In the following, assume that the formula split is viable for the respectively used Boolean connective σ and that φ is split into the independent subformulas $\{\varphi_1, \varphi_2\}$. Further, let I_1 and I_2 be the algebraic intervals resulting from processing the cylindrical algebraic coverings algorithm for φ_1 and φ_2, respectively. We denote the satisfiability of an interval using T and likewise the unsatisfiability of an interval as F.

I_1	I_2	$\varphi_1 \wedge \varphi_2$
T	T	$I_1 \cap I_2$
T	F	I_2
F	T	I_1
F	F	$I_1 \cup I_2$

I_1	I_2	$\varphi_1 \vee \varphi_2$
T	T	$I_1 \cup I_2$
T	F	I_2
F	T	I_1
F	F	$I_1 \cap I_2$

Definition 4.1.1 shows how the result of two independent subformulas can be combined. This can naturally be extended when we have more than two independent subformulas.

In the following, we will present the adapted version of the cylindrical algebraic coverings algorithm. For this, we first must define how to compute the intersection of two algebraic intervals when necessary. This is presented in Algorithm 12. We use the notation $\bigcap_{I \in A} I := \texttt{merge}(A)$ to denote the computation of the intersection of a set of intervals A. The algorithm $\texttt{merge}(A)$ takes a set of algebraic intervals A as input and returns another algebraic interval I, which is exactly the intersection of all intervals in A. The lower and upper bounds I_l and I_u of the resulting algebraic interval I are computed as the minimum and maximum of the lower and upper bounds of the intervals in A, respectively. The sets of polynomials I_L and I_U defining these boundaries are the same as in the algebraic interval, with the smallest lower bound and the largest upper bound, respectively. These sets are updated whenever a new interval with a smaller lower bound or a larger upper bound is found. The sets I_{P_i} and $I_\perp$ are constructed as the union of the sets I'_{P_i} and $I'_\perp$ of intervals $I' \in A$. This ensures that the resulting interval I is as tight as possible and fully defined.

Algorithm 12: `merge`(A)

Input: A set of algebraic intervals A
Output: An algebraic interval I, which is the intersection of all intervals in A

```
1  Initialize I := I_l = ∞, I_u = −∞, I_L = ∅, I_U = ∅, I_{P_i} = ∅, I_⊥ = ∅
2  foreach I' ∈ A do
3  |  Extract I'_l, I'_u, I'_L, I'_U, I'_{P_i}, I'_⊥ from I'
4  |  if I'_l < I_l then
5  |  |  I_l := I'_l
6  |  |  I_L := I'_L
7  |  if I'_u > I_u then
8  |  |  I_u := I'_u
9  |  |  I_U := I'_U
10 |  I_{P_i} := I_{P_i} ∪ I'_{P_i}
11 |  I_⊥ := I_⊥ ∪ I'_⊥
12 Define Interval I := (I_l, I_u, I_L, I_U, I_{P_i}, I_⊥)
```

Output: I

As a counterpart to the merge operation, where we calculate the intersection of a set of intervals, we also need to calculate the union of a set of intervals. We have introduced the notation $\bigcup_{I \in A} I$ for this. Instead of merging the set of intervals into a single one, we will view the intervals separately, adding each interval individually. To define the union operation, we must make adaptions to Algorithms 1–4. The changes are shown in Algorithms 13, 14, 16 and 17. One notable change is that the formula φ is no longer global and is passed as an argument. By passing the formula φ as an argument, we can change which formula to process by the respective algorithms. The prefix $Q_1 x_1, \ldots, Q_n x_n$, which defines the variable ordering and quantification, is still global. The return value of Algorithms 2–4, are tuples of the form $(\texttt{SAT}, I)$ or $(\texttt{UNSAT}, I)$, where I is an algebraic interval. This is changed to $(\texttt{SAT}, \{I_1, \ldots I_m\})$ or $(\texttt{UNSAT}, \{I_1, \ldots I_m\})$ where for every $I \in \{I_1, \ldots I_m\}$ it holds that $s \times I$ can or can not be extended to be a satisfying model for $\bar{\varphi}$ for any $s_i \in I$. Thus, instead of returning a single interval, we return a set of intervals. This is necessary because the result of processing a split formula may be two or more intervals instead of just one. In the following, we will present the adapted formulas in detail.

Algorithm 13: `user_call_split`(φ)

Input: A formula φ
Data: Global prefix $Q_1 x_1, \ldots, Q_n x_n$ and a Boolean flag $early\_return \in \{True, False\}$
Output: Either `SAT` or `UNSAT`

```
1  (f, O) := recourse_split(φ, ())          // Algorithm 14
2  return f
```

Algorithm 13 is the user interface for the CAlC algorithms. The change to Algorithm 1 is that the formula φ is not global anymore but passed as an argument. Further, a Boolean flag $early\_return \in \{True, False\}$ is introduced. This flag is used in later algorithms. In Line 1 Algorithm 14 is called, which is the adaption

of Algorithm 2. The formula φ and the Boolean flag *early_return* are passed as arguments. The sample point $s = ()$ is also passed as an argument.

Algorithm 14: `recourse_split`(φ, s)

Data: Global prefix $Q_1 x_1, \ldots, Q_n x_n$ and a Boolean flag *early_return* $\in \{True, False\}$
Input: Formula φ, a sample point $s = (s_1, \ldots, s_{i-1}) \in \mathbb{R}^{i-1}$
Output: Either $(\texttt{SAT}, \{I_1, \ldots I_m\})$ or $(\texttt{UNSAT}, \{I_1, \ldots I_m\})$

```
1  Let σ such that φ = φ_1 σ ... σ φ_m
2  Φ_split = {φ}                                          // Init split value
3  if (σ = ∨ and Q_i = ∃) or (σ = ∧ and Q_i = ∀) then
4  |  Φ_split := formula_split({φ_1, ..., φ_m}, x_i, s)   // Algorithm 11
5  if |Φ_split| = 1 then                                  // No split is possible
6  |  if Q_i = ∃ then
7  |  |  return exists_split(φ, s)                       // Algorithm 16
8  |  else
9  |  |  return forall_split(φ, s)                       // Algorithm 17
10 // Formula is split in two or more independent subformulas
11 (I_sat, I_unsat) := get_results(Φ, s, σ, return_early) // Algorithm 15
12 if σ = ∨ then
13 |  if I_sat ≠ ∅ then                                   // There are true results
14 |  |  return (SAT, I_sat)
15 |  return (UNSAT, {merge(I_unsat)})                    // Algorithm 12
16 else // σ = ∧
17 |  if I_unsat ≠ ∅ then                                 // There are false results
18 |  |  return (UNSAT, I_unsat)
19 |  return (SAT, {merge(I_sat)})                       // Algorithm 12
```

In Line 1, we initialize σ with the Boolean connective of the formula φ, i.e. if $\varphi = \varphi_1 \wedge \ldots \wedge \varphi_m$, then $\sigma = \wedge$. With this information, together with the quantifier Q_i of the variable x_i, which is currently of concern, we can determine if the split of the current formula φ is viable. In any case, in Line 2, we initialize the set of formulas Φ_{split} with $\{\varphi\}$. We do this to simplify the following lines of the algorithm and to avoid redundancy. In Line 3, we check if a formula split is viable according to Definition 4.0.3. If that is the case we call Algorithm 11 in Line 4 to calculate $\Phi_{split} := formula\_split(\{\varphi_1, \ldots, \varphi_m\}, x_i, s)$ following Definition 4.0.2. After, in Line 5, we check if the formula was split into two or more independent subformulas. If that is not the case, i.e., $|\Phi_{split}| = 1$, then the formula φ, for any reason, could not be split into multiple independent subformulas. Then, in Lines 6–9 we call Algorithm 16 or Algorithm 17 according to the quantifier Q_i of the variable x_i which is currently of concern. If the formula φ is split into multiple independent subformulas, we handle the two possible cases differently based on the logical connective σ. If $\sigma = \vee$, then we handle the disjunction in Lines 12–15. If $\sigma = \wedge$, then we handle the conjunction in Lines 16–19. In any case, in Line 11, we call Algorithm 15 to obtain the sets $\mathbb{I}_{sat}$ and $\mathbb{I}_{unsat}$ which contain the resulting algebraic intervals of the satisfiable and unsatisfiable formulas in Φ, respectively. Then, depending on the sets $\mathbb{I}_{sat}$ and $\mathbb{I}_{unsat}$, the functions differ.

When handling the split subformulas of a disjunction, it suffices to check if there

is at least one satisfiable subformula to determine the satisfiability of the formula φ passed as input. If that is the case, then the whole disjunction is satisfiable. That is the case, as the combined formula of Φ is of the form $\varphi_1 \vee \ldots \vee \varphi_m$ with $\varphi_i \in \Phi$. Thus, if there exists at least one subformula $\varphi_i \in \Phi$ such that s can be extended to a satisfying model for φ_i, this same model is also a satisfying model for the combined formula $\varphi_1 \vee \ldots \vee \varphi_m$. This also implies that if we set the flag *return_early* to *True*, then the algorithm already returns if there is at least one satisfiable subformula, which can have the effect that some formulas in Φ do not have to be processed at all. Thus, in Line 13, we check if $\mathbb{I}_{sat} \neq \emptyset$. We return $(\texttt{SAT}, \mathbb{I}_{sat})$ if that is the case. This way, if there is more than one calculated satisfiable result in $\mathbb{I}_{sat}$, we return all of them. This means that the respective algebraic intervals are processed separately in the following steps. Intuitively, this is because any of the satisfiable subformulas in Φ can be used to extend s to a satisfying model for the combined formula $\varphi_1 \vee \ldots \vee \varphi_m$. Thus, they can all be used to generalize s individually. If there are no satisfiable subformulas in Φ, then we return $(\texttt{UNSAT}, \{\texttt{merge}(\mathbb{I}_{unsat})\})$. This means the whole disjunction is unsatisfiable if there is no satisfiable subformula in Φ. In other words, if there does not exist a subformula $\varphi_i \in \Phi$ such that s can be extended to a satisfying model for φ_i, then s can not be extended to a satisfying model for the combined formula $\varphi_1 \vee \ldots \vee \varphi_m$. We combine the resulting algebraic intervals in $\mathbb{I}_{unsat}$ into a single algebraic interval using Algorithm 12 and return $(\texttt{UNSAT}, \{\texttt{merge}(\mathbb{I}_{unsat})\})$. This way, the resulting generalization of s is as tight as possible.

The logic of processing the split subformulas of a conjunction is similar. There, it suffices to check if there is at least one unsatisfiable subformula to determine the satisfiability of the input formula φ. If that is the case, then the whole conjunction is unsatisfiable. That is the case, as the combined formula of Φ is of the form $\varphi_1 \wedge \ldots \wedge \varphi_m$ with $\varphi_i \in \Phi$. Thus, if there exists at least one subformula $\varphi_i \in \Phi$ such that s can not be extended to a satisfying model for φ_i, then s can not be extended to a satisfying model for the combined formula $\varphi_1 \wedge \ldots \wedge \varphi_m$. This also implies that if we set the flag *return_early* to *True*, then the algorithm already returns if there is at least one unsatisfiable subformula, which can have the effect that some formulas in Φ do not have to be processed at all. Thus, in Line 17, we check if $\mathbb{I}_{unsat} \neq \emptyset$. We return $(\texttt{UNSAT}, \mathbb{I}_{unsat})$ if that is the case. This way, we return all of them if there is more than one calculated unsatisfiable result in $\mathbb{I}_{unsat}$. This means that the respective algebraic intervals are processed separately in the following steps. Intuitively, this is because any of the unsatisfiable subformulas in Φ can be used to show that s can not be extended to a satisfying model for the combined formula $\varphi_1 \wedge \ldots \wedge \varphi_m$. Thus, they can all be used to generalize s individually. If there are no unsatisfiable subformulas in Φ, then we return $(\texttt{SAT}, \{\texttt{merge}(\mathbb{I}_{sat})\})$. If there is no unsatisfiable subformula in Φ, then the whole conjunction is satisfiable. In other words, if there does not exist a subformula $\varphi_i \in \Phi$ such that s can not be extended to a satisfying model for φ_i, then s can be extended to a satisfying model for the combined formula $\varphi_1 \wedge \ldots \wedge \varphi_m$. We combine the resulting algebraic intervals in $\mathbb{I}_{sat}$ into a single algebraic interval using Algorithm 12 and return $(\texttt{SAT}, \{\texttt{merge}(\mathbb{I}_{sat})\})$. This way, the resulting generalization of s is as tight as possible.

Algorithm 15: `get_results`(Φ, s, σ)

Data: Global prefix $Q_1 x_1, \ldots, Q_n x_n$ and a Boolean flag $return\_early \in \{True, False\}$
Input: Set of formulas Φ, a sample point $s = (s_1, \ldots, s_{i-1}) \in \mathbb{R}^{i-1}$ and Boolean connective $\sigma \in \{\vee, \wedge\}$
Output: Tuple of Sets $(\mathbb{I}_{sat}, \mathbb{I}_{unsat})$

```
1  I_sat = ∅                                          // Initialize SAT results
2  I_unsat = ∅                                        // Initialize UNSAT results
3  foreach φ ∈ Φ do
4  |  if Q_i = ∃ then
5  |  |  (f, I) := exists_split(φ, s, early_return)          // Algorithm 16
6  |  else // Q_i = ∀
7  |  |  (f, I) := forall_split(φ, s, early_return)          // Algorithm 17
8  |  if f = SAT then
9  |  |  I_sat = I_sat ∪ I
10 |  |  if return_early and σ = ∨ then
11 |  |  |  return (I_sat, I_unsat)
12 |  else
13 |  |  I_unsat = I_unsat ∪ I
14 |  |  if return_early and σ = ∧ then
15 |  |  |  return (I_sat, I_unsat)
16 return (I_sat, I_unsat)
```

In Algorithm 15 we initialize two sets $\mathbb{I}_{sat}$ and $\mathbb{I}_{unsat}$ in Lines 1–2. These sets are used to store the resulting algebraic intervals of the respectively satisfiable and unsatisfiable formulas in Φ. Then, in Line 3, we iterate over the formulas $\varphi \in \Phi$ to obtain the results. Depending on the quantifier Q_i of the variable that is currently of concern, we call either Algorithm 16 or Algorithm 17 in Lines 5–7. While the variable x_i itself is not given as an argument, one can deduce the index i from the sample point $s = (s_1, \ldots, s_{i-1}) \in \mathbb{R}^{i-1}$. The result of the call to either Algorithm 16 or Algorithm 17 is a tuple $(f, \mathbb{I})$ where f is either `SAT` or `UNSAT` and $\mathbb{I}$ is a set of algebraic intervals. Based on the value of f, we add the intervals in $\mathbb{I}$ to the respective set $\mathbb{I}_{sat}$ or $\mathbb{I}_{unsat}$. Further, based on if the flag $early\_return$ is set to $True$ and the Boolean connective σ of the formula φ, we check if we can return early. If $return\_early = True$ and $\sigma = \vee$ and there is a formula $\varphi \in \Phi$ such that s can be extended to a satisfying model for the given φ, then the algorithm returns early. Analogously, if $return\_early = True$ and $\sigma = \wedge$ and there is a formula $\varphi \in \Phi$ such that s can not be extended to a satisfying model for the given φ, then the algorithm returns early. If $return\_early$ is set to $False$ or any of the cases for an early return does not happen, then we continue with the next formula in Φ. When either all formulas in Φ are processed, or a case where we can return early occurs, we return the tuple $(\mathbb{I}_{sat}, \mathbb{I}_{unsat})$. Further, as the formulas in Φ are pairwise independent, the order in which we process the formulas does not matter. Thus, one can sort the formulas by complexity according to some metric or process them in parallel.

Algorithm 16: `exists_split`(φ, s)

Data: Global prefix $Q_1 x_1, \ldots, Q_n x_n$ and a Boolean flag $early\_return \in \{True, False\}$
Input: Formula φ currently of interest, sample point $s = (s_1, \ldots, s_{i-1}) \in \mathbb{R}^{i-1}$
Output: $(\texttt{SAT}, \{I_1, \ldots I_m\})$ or $(\texttt{UNSAT}, \{I_1, \ldots I_m\})$ where for every $I \in \{I_1, \ldots I_m\}$ it holds that $s \times I$ can or can not be extended to be a satisfying model for $\bar{\varphi}$ for any $s_i \in I$. In any case, the algebraic information attached to I describes how s can be generalized.

$\mathbb{I}_{unsat} := \emptyset$
while $\bigcup_{I \in \mathbb{I}_{unsat}} I \neq \mathbb{R}$ **do**
  $s_i :=$ `sample_outside`$(\mathbb{I}_{unsat})$
  if $\bar{\varphi}[s \times s_i]$ = False **then**
    $(f,O) :=$ (`UNSAT`, {`get_enclosing_interval`(s,s_i)})
    `// Algorithm 5`
  else if $\bar{\varphi}[s \times s_i]$ = True **then**
    $(f,O) :=$ (`SAT`, {`get_enclosing_interval`(s,s_i)}) `// Algorithm 5`
  else
    $(f,O) :=$ `recourse_split`$(\varphi, s \times s_i)$
    `// Algorithm 14, recursive call`
  if $f = SAT$ **then**
    $\mathbb{I}_{sat} := \emptyset$
    foreach $I_O \in O$ **do**
      $R :=$ `characterize_interval`(s, I_O) `// Algorithm 6`
      $I :=$ `interval_from_characterization`$((s_1, \ldots, s_{i-2}), s_{i-1}, R)$
      `// Algorithm 8`
      $\mathbb{I}_{sat} = \mathbb{I}_{sat} \cup \{I\}$
    return $(SAT, \mathbb{I}_{sat})$
  $\mathbb{I}_{unsat} := \mathbb{I}_{unsat} \cup O$ `//` $f =$ `UNSAT`
$R :=$ `characterize_covering`$(s, \mathbb{I}_{unsat})$ `// Algorithm 7`
$I :=$ `interval_from_characterization`$((s_1, \ldots, s_{i-2}), s_{i-1}, R)$
`// Algorithm 8`
return $(UNSAT, \{I\})$

Algorithm 17: `forall_split`(φ, s)

Data: Global prefix $Q_1x_1, \ldots, Q_nx_n$ and a Boolean flag $early\_return \in \{True, False\}$
Input: Formula φ currently of interest, sample point $s = (s_1, \ldots, s_{i-1}) \in \mathbb{R}^{i-1}$
Output: $(\texttt{SAT}, \{I_1, \ldots I_m\})$ or $(\texttt{UNSAT}, \{I_1, \ldots I_m\})$ where for every $I \in \{I_1, \ldots I_m\}$ it holds that $s \times I$ can or can not be extended to be a satisfying model for $\bar{\varphi}$ for any $s_i \in I$. In any case, the algebraic information attached to I describes how s can be generalized.

```
1  I_sat := ∅
2  while ⋃_{I∈I_sat} I ≠ ℝ do
3      s_i := sample_outside(I_sat) if φ̄[s × s_i] = False then
4          (f,O) := (UNSAT, {get_enclosing_interval(s,s_i)})
                // Algorithm 5
5      else if φ̄[s × s_i] = True then
6          (f,O) := (SAT, {get_enclosing_interval(s,s_i)}) // Algorithm 5
7      else
8          (f,O) := recourse_split(φ, s × s_i, early_return)
                // Algorithm 14, recursive call
9      if f = UNSAT then
10         I_unsat = ∅
11         foreach I_O ∈ O do
12             R := characterize_interval(s, I_O)          // Algorithm 6
13             I :=
                   interval_from_characterization((s_1, ..., s_{i-2}), s_{i-1}, R)
                   // Algorithm 8
14             I_unsat = I_unsat ∪ {I}
15         return (UNSAT, I_unsat)
16     I_sat := I_sat ∪ O                                  // f = SAT
17 R := characterize_covering(s, I_sat)                    // Algorithm 7
18 I := interval_from_characterization((s_1, ..., s_{i-2}), s_{i-1}, R)
     // Algorithm 8
19 return (SAT, {I})
```

Now we will present the adapted versions of Algorithms 3 and 4. The changes are shown in respectively Algorithms 16 and 17. Both algorithms are adapted in the same way. Thus, we are going to go over the changes only once. The main change is that the return value is now a tuple of the form $(\texttt{SAT}, \{I_1, \ldots I_m\})$ or $(\texttt{UNSAT}, \{I_1, \ldots I_m\})$ where for every $I \in \{I_1, \ldots I_m\}$ it holds that $s \times I$ can or can not be extended to be a satisfying model for $\bar{\varphi}$ for any $s_i \in I$. In any case, the algebraic information attached to I describes how s can be generalized. Further, the formula φ is not global anymore but passed as an argument, enabling the processing of each independent subformula individually. The formula prefix $Q_1x_1, \ldots, Q_nx_n$, which defines the variable ordering and quantification, is still global. Another change is that a Boolean flag $early\_return \in \{True, False\}$ is also passed as an argument. This flag indicates whether the algorithm should return early if a formula already decides the result. Another change is that the tuple (f, O) no longer contains a single interval I but a set of intervals O. That is, in both algorithms, in Lines 4 and 6 we

call Algorithm 5 to compute the satisfiability invariant around $s \times s_i$ for the given formula φ. The return value of Algorithm 5 is a single algebraic interval I, from which we initialize a set of intervals $O := \{I\}$. This change is necessary as in Line 8, in both Algorithm 16 and Algorithm 17, we call Algorithm 14 recursively. The return value of Algorithm 14 is a tuple (f, O), where f is either `SAT` or `UNSAT` and O is a set of intervals. So in total, we have the tuple (f, O) where f is either `SAT` or `UNSAT` and O is a set of algebraic intervals in any case. As a consequence of this change, we must also adapt how we handle these intervals in this tuple. This inherently is based on f for the respective algorithm. In Algorithm 16, if $f = \texttt{SAT}$, then we need to characterize the intervals in O and return them as a set of intervals $\mathbb{I}_{sat}$. In Line 11, we initialize the set $\mathbb{I}_{sat}$ with the empty set. Then, in Line 12, we iterate over the intervals $I_O \in O$ and call Algorithm 6 and Algorithm 8 for each interval I_O and add the resulting algebraic interval I to $\mathbb{I}_{sat}$. Finally, in Line 16, we return $(\texttt{SAT}, \mathbb{I}_{sat})$. Analogously, in Algorithm 17, if $f = \texttt{UNSAT}$, then we need to characterize the intervals in O and return them as a set of intervals $\mathbb{I}_{unsat}$. In Line 10, we initialize the set $\mathbb{I}_{unsat}$ with the empty set. Then, in Line 11, we iterate over the intervals $I_O \in O$ and call Algorithm 6 and Algorithm 8 for each interval I_O and add the resulting algebraic interval I to $\mathbb{I}_{unsat}$. Finally, in Line 15, we return $(\texttt{UNSAT}, \mathbb{I}_{unsat})$. In both cases, this also ensures that the return value is of the form $(f, \{I_1, \dots\})$ where f is either `SAT` or `UNSAT` and $I_1, \dots$ are algebraic intervals. In the other case, for both algorithms, meaning if $f = \texttt{SAT}$ in Algorithm 16 or $f = \texttt{UNSAT}$ in Algorithm 17, we need to add the intervals in O to the set of intervals $\mathbb{I}_{unsat}$ or $\mathbb{I}_{sat}$, respectively.

Chapter 5

Variable Orderings

In the following section, we present how different variable orderings can be used to improve the performance of the coverings method. It is a well-known fact that the performance of the CAD algorithm and thus also of the CAlC algorithm is heavily dependent on the variable ordering used [EF19, LXZZ21, HEW$^+$14, DSS04]. The underlying theory of this thesis is the theory of non-linear real arithmetic. In this theory, the order of differently quantified variables is of importance for the semantics of the formula. We can not arbitrarily change the order of the variables, as this would inherently change the semantics and, thus, the result when solving the quantifier elimination problem or the decision problem. In the following, we are going to define the notion of admissible variable orderings, which are variable orderings that preserve the semantics of the formula.

Definition 5.0.1 (Quantifier Block)**.**
Let $\varphi := Q_{k+1}x_{k+1}, \dots, Q_n x_n.\bar{\varphi}$ be a formula in prenex normal form with $Q_i \in \{\forall, \exists\}$. Then the variables $\{x_1, x_k\} = B_0$ are the free variables. Without loss of generality, we can rewrite the prefix of φ equivalently using quantifiers blocks by combining the variables of identical subsequent quantifiers into sets. Thus, we can rewrite the prefix of φ as $B_0.Q_1B_1 \dots Q_mB_m$. Where B_i are pairwise disjoint sets of variables and we have $Q_i \neq Q_{i+1}$ for all $i \in \{1, \dots, m\}$. The sets Q_iB_i are called quantifier blocks.

A quantifier block is a set of adjacent variables quantified by the same quantifier. Thus, the variables inside a quantifier block are either existentially quantified, universally quantified, or free. We add the free variables to the first quantifier block B_0. This means the free variables are always at the beginning of any variable ordering.

Definition 5.0.2 (Admissible Ordering)**.**
Let $\varphi := B_0.Q_1B_1 \dots Q_mB_m.\bar{\varphi}$ be a formula in prenex normal form with quantifier blocks as defined in Definition 5.0.1. Then a variable ordering $\prec$ is called admissible for φ if it holds that $x_i \prec x_j \implies x_i \in B_a \wedge x_j \in B_b$ with $a \leq b$.

We call a variable ordering admissible, as defined in Definition 5.0.2 if the order of the variables is preserved among quantifier blocks. We observe that we can arbitrarily reorder the variables within such a quantifier block without changing the semantics of the formula. Thus, a variable ordering may change the ordering of variables within a single quantifier block but not between quantifier blocks. In the following, all variable orderings of interest are admissible. This means we always assume that the order of

quantifier blocks is preserved. The amount of admissible variable orderings is bound by the size of the given quantifier blocks, as we can only change the order of variables within a single quantifier block. The amount of possible variable orderings is given by $\prod_{i=0}^{m} \max(1,|B_i|)$.

Example 5.0.1 (Admissible and non-admissible variable ordering)**.**
Let $\varphi := \exists x_1.\forall x_2.\forall x_3.x_1 + x_2 = x_3$. This formula is unsatisfiable. Further, it is in prenex normal form with quantifier blocks $B_0 = \emptyset, B_1 = \{x_1\}$ and $B_2 = \{x_2, x_3\}$. There are $\prod_{i=0}^{2} \prod_{i=0}^{m} \max(1,|B_i|) = 1 \cdot 1 \cdot 2 = 2$ possible admissible variable orderings. The semantics of formula φ are given by the variable ordering $x_1 \prec x_2 \prec x_3$. The semantics remain the same for the admissible variable ordering $x_1 \prec x_3 \prec x_2$. This variable ordering is admissible, as only the order of x_2 and x_3 has changed within the quantifier block B_2.

Now, assume that the variable ordering $x_2 \prec x_1 \prec x_3$ is used. This variable ordering is non-admissible, as the order of the variables x_1, x_2 has changed and these variables are in different quantifier blocks. The resulting formula $\varphi' := \exists x_2.\forall x_1.\forall x_3.x_1 + x_2 = x_3$ given by this variable ordering is satisfiable.

As shown in Example 5.0.1, the change in the order of quantification can impact the satisfiability of the given formula. The formula semantics are thus preserved by admissible variable orderings but not by non-admissible variable orderings.

5.1 Variable Orderings in Literature

In the following, we present two variable orderings that have been used in literature [HEW+14, DSS04, EBDW14]. Note that, in order to obtain admissible variable orderings, as defined in Definition 5.0.2, the following orderings are restricted to the variables in the quantifier blocks. This means that each quantifier block is ordered separately according to the given criteria, but the order of quantifier blocks is preserved. By doing this, we ensure that the variable ordering does not change the semantics of the formula.

Definition 5.1.1 (Triangular Variable Ordering [EBDW14])**.**
Let φ be a formula, P the set of polynomials defined by φ and V the set of variables defined by φ. Then, the triangular variable ordering is defined as follows: The variable ordering is chosen according to the following criteria, starting with the first and breaking ties with successive ones:

1. Let $v^{[1]} = \max\{\mathbf{deg}(f,v) | f \in P\}$. Then $x \prec y$ if $y^{[1]} < x^{[1]}$.
2. Let $v^{[2]} = \max\{\mathbf{tdeg}(\mathbf{lcoeff}(f,v)) | f \in P, f \text{ contains } v\}$. Then $x \prec y$ if $y^{[2]} < x^{[2]}$.
3. Let $v^{[3]} = \sum_{f \in P} \mathbf{deg}(f,v)$. Then $x \prec y$ if $y^{[3]} < x^{[3]}$.

Definition 5.1.2 (Brown Variable Ordering [HEW+14])**.**
Let φ be a formula, P the set of polynomials defined by φ and V the set of variables defined by φ. Then, the brown variable ordering is defined as follows: The variable ordering is chosen according to the following criteria, starting with the first and breaking ties with successive ones:

1. Then set $x \prec y$ if $y^{[1]} < x^{[1]}$.

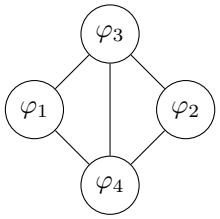
2. Let $v^{[4]} = \max\{\mathbf{tdeg}(t) | t \text{ is monomial from } f \in P, t \text{ contains } v\}$. Then $x \prec y$ if $y^{[4]} < x^{[4]}$.

3. Let $v^{[5]} = \#\{t | t \text{ is monomial from } f \in P, t \text{ contains } v\}$. Then $x \prec y$ if $y^{[5]} < x^{[5]}$.

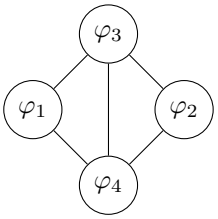
5.2 Earliest Split Variable Ordering

In the following, we are going to present a heuristical approach to split a given formula φ into independent subformulas with as few variables assigned as possible. To see any difference between the different variable orderings, we refer to Example 5.2.1, where we show that the variable ordering impacts the resulting split.

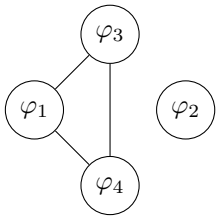
Example 5.2.1 (Formula splits for different variable orderings)**.**
Assume that the formula φ has the subformulas $\Phi := \{\varphi_1(x_5,x_6), \varphi_2(x_1,x_2,x_3), \varphi_3(x_1,x_2,x_3,x_4,x_5), \varphi_4(x_1,x_2,x_3,x_4,x_5)\}$. Further we assume that $\sigma := \wedge$, but the same holds for $\sigma := \vee$. For simplicity, we assume that no unassigned variable vanishes when evaluating a formula φ_i at a sample point s. First, we assume we have the variable ordering $x_1 \prec x_2 \prec x_3 \prec x_4 \prec x_5$. We will now show the different Graphs built by Algorithm 11 for different sample points s. For an explanation of how to build these Graphs, we refer to Algorithm 11 and Example 4.0.4.



(a) Graph for $s = ()$



(b) Graph for $s = (x_1 = 0)$



(c) Graph for $s = (x_1 = 0, x_2 = 0)$

Figure 5.1: Graphs for different sample points s for the variable ordering $x_1 \prec x_2 \prec x_3 \prec x_4 \prec x_5$

We can see that for $s = ()$ and $s = (x_1 = 0)$, the Graphs are the same, and there is exactly one connected component in the Graphs as shown in Figures 5.1a and 5.1b. Thus, the given formula set φ can not be split into two or more independent subformulas for these sample points. However, for $s = (x_1 = 0, x_2 = 0)$ the Graph has two connected components as shown in Figure 5.1c. Thus, the given formula set φ can be split into two independent subformulas for this sample point. Thus, we must assign at least two variables to obtain two independent subformulas.

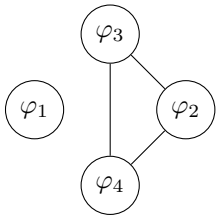


Figure 5.2: Graph for $s = ()$, for the variable ordering $x_5 \prec x_1 \prec x_2 \prec x_3 \prec x_4$

Now, consider the same formula set φ but with the variable ordering $x_5 \prec x_1 \prec x_2 \prec x_3 \prec x_4$. For $s = ()$, the graph has two connected components as shown in Figure 5.2. Thus, the given formula set φ can be split into two independent subformulas for this sample point. Accordingly, we do not need to assign a single variable to obtain two independent subformulas.

For an unspecified formula, we show that the first possibility to split the formula into two or more independent subformulas occurs at different sample points for two different variable orderings. The first possibility for a split for one variable ordering happens when 3 variables are assigned in s. For the other variable ordering, it happens when no variable is assigned in s.

In the following, we are interested in a heuristical algorithm to choose the variable ordering, such that the split of the original formula φ into two or more independent subformulas happens with the least amount of assigned variables possible. To do this, we will extend and adapt the definition of the graph $G = (V, E)$ as introduced for Algorithm 11. Given a set of formulas Φ, a variable x_i and a sample $s = (s_1, \ldots, s_{i-1}) \in \mathbb{R}^{i-1}$ we define the graph G as follows:

- The vertices of the graph are exactly the formulas in Φ.
- Edge between two vertices φ, φ' iff φ and φ' are not independent with respect to x_i and s.

We extend this definition by adding a labeling function $l : V \times V \to \mathcal{P}(\mathbb{X})$ to the graph G. Where $\mathbb{X} := \bigcup_{\varphi \in \Phi} \texttt{vars}(\varphi)$ is the set of all used variables in the formulas in Φ and $\mathcal{P}$ denotes the powerset. The labeling function l is defined as follows:

- $l(\varphi_i[s], \varphi_j[s]) = \texttt{vars}(\varphi_j) \cap \texttt{vars}(\varphi_j)$, where $\texttt{vars}(\varphi_i)$ denotes the set of variables used in φ_i and s is a (partial) sample point.

Further, we are going to introduce the following invariant that holds for the graph G:

- For any two vertices $\varphi_i, \varphi_j \in V$ there exists an edge $(\varphi_i, \varphi_j) \in E$ iff $l(\varphi_i, \varphi_j) \neq \emptyset$.

Example 5.2.2 (Formula Split Graph with edge labeling function)**.**
Assume that the formula φ has the subformulas $\Phi := \{\varphi_1(x_5,x_6), \varphi_2(x_1, x_2, x_3), \varphi_3(x_1, x_2, x_3, x_4, x_5), \varphi_4(x_1, x_2, x_3, x_4, x_5)\}$. For simplicity, we assume that no unassigned variable vanishes when evaluating a formula φ_i at a sample point s. Let $\sigma \in \{\wedge, \vee\}$ be an Boolean connective and $s = ()$. Then, the graph $G = (V,E)$ and the edge labeling function l is defined as follows:

- $V = \{\varphi_1, \varphi_2, \varphi_3, \varphi_4\}$

- $l(\varphi_1, \varphi_2) = \emptyset$
- $l(\varphi_1, \varphi_3) = \{x_5\}$
- $l(\varphi_1, \varphi_4) = \{x_5\}$
- $l(\varphi_2, \varphi_3) = \{x_1, x_2, x_3\}$
- $l(\varphi_2, \varphi_4) = \{x_1, x_2, x_3\}$
- $l(\varphi_3, \varphi_4) = \{x_1, x_2, x_3, x_4, x_5\}$
- $E = \{(\varphi_1, \varphi_3), (\varphi_1, \varphi_4), (\varphi_2, \varphi_3), (\varphi_2, \varphi_4), (\varphi_3, \varphi_4)\}$

Which results in the graph G as shown in Figure 5.3.

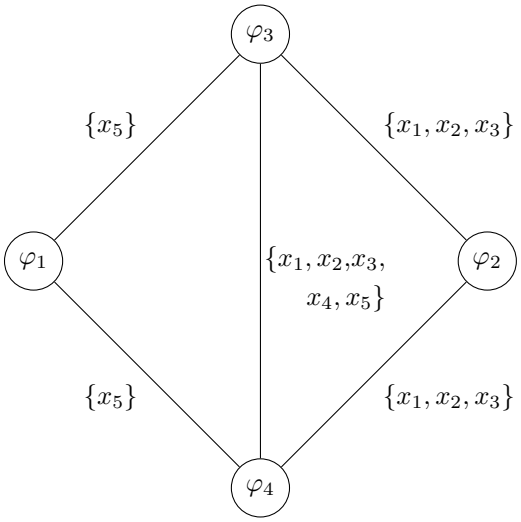


Figure 5.3: Graph $G = (V,E)$ for $s = ()$ and the labeling function l

And for $s = (x_1 = 0, x_2 = 0)$ the graph $G = (V,E)$ and the edge labeling function l is defined as follows:

- $V = \{\varphi_1, \varphi_2, \varphi_3, \varphi_4\}$
- $l(\varphi_1, \varphi_2) = \emptyset$
- $l(\varphi_1, \varphi_3) = \{x_5\}$
- $l(\varphi_1, \varphi_4) = \{x_5\}$
- $l(\varphi_2, \varphi_3) = \{x_3\}$
- $l(\varphi_2, \varphi_4) = \{x_3\}$
- $l(\varphi_3, \varphi_4) = \{x_3, x_4, x_5\}$
- $E = \{(\varphi_1, \varphi_3), (\varphi_1, \varphi_4), (\varphi_2, \varphi_3), (\varphi_2, \varphi_4), (\varphi_3, \varphi_4)\}$

Which results in the graph G as shown in Figure 5.4.

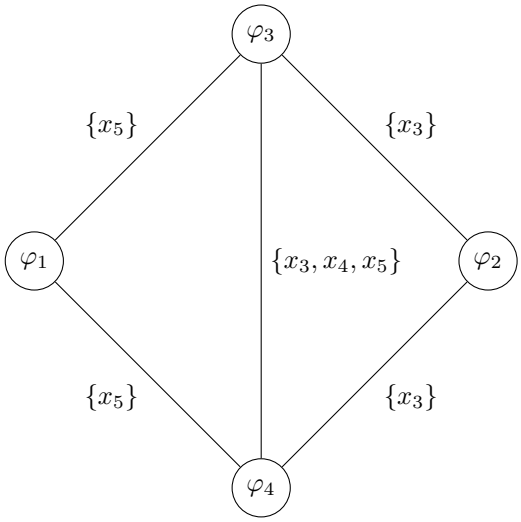


Figure 5.4: Graph $G = (V,E)$ for $s = (x_1 = 0, x_2 = 0)$ and the labeling function l

Now we are interested in a variable ordering heuristic, with restrain to the given global quantifier blocks, as defined in Definition 5.0.2, that results in a variable ordering such that a formula split into two or more independent subformulas is achieved with the least amount of variables assigned as possible. From the definition of the graph G and with the same arguments as presented in Algorithm 11, we know that the connected components exactly correspond to the independent subformulas of Φ. Given the graph G and the labeling function l as defined above for a given formula φ and we want to assign the variable x_i, then we need to remove any occurance of x_i from the sets of variables of the labeling function l. If, after removing the variable x_i from the labeling function, and it holds that $l(\varphi_i, \varphi_j) = \emptyset$ then we need to remove the edge (φ_i, φ_j) from the set of edges in G if it exists. Formally, we want to create a variable ordering, such that when the variables are assigned in that given order, the graph G contains two or more connected components with the least amount of assigned variables possible. Visually, when inspecting the graph shown in Figure 5.3 we can see that an optimal strategy is to assign x_5 first because then the graph already forms two connected components. However, this decision is constrained by the quantifier blocks of the given formula. Meaning that if we have $\exists\{x_1,x_2\}.\forall\{x_3, x_4, x_5\}$ as the quantifier blocks, then we can not assign x_5 first, because x_5 is not in the first quantifier block. Thus, we need to assign the variables x_1, x_2 first in any case. Instead of computing the optimal variable ordering for a given formula φ, we will present a heuristic to compute a variable ordering in Algorithm 18.

Algorithm 18: `early_split_var_order`(φ)

Data: Global prefix $Q_1x_1, \ldots, Q_nx_n$
Input: Formula φ.
Output: A list variable of variables representing the ordering $x_i \prec \ldots \prec x_j$

$Q_{blocks} := Q_1\{x_1, \ldots, x_i\}.Q_2\{x_{i+1}, \ldots\}.\ldots$
// Quantifier blocks Definition 5.0.1
$Ordering = []$ // Initialize empty list
Compute $G = (V,E)$ and labeling function l
foreach Q_{block} *in* Q_{blocks} **do** // Iterate over quantifier blocks in order
  $Var_{block} := \{x \in Q_{block} \mid x \in \texttt{vars}(\varphi)\}$ // Variables in block
  while $Var_{block} \neq \emptyset$ **do**
    $var_{poss} := \arg\max_{x \in Var_{block}} \sum_{\varphi_i, \varphi_j \in \Phi} |l(\varphi_i, \varphi_j) \cap \{x\}|$
    // Choose variable occurring the most often in $l(\varphi_i, \varphi_j)$
    if $|var_{poss}| > 1$ **then**
      $var_{poss} := \arg\min_{x \in var_{poss}} \sum_{\varphi_i, \varphi_j \in \Phi} |l(\varphi_i, \varphi_j) \cap \{x\}|$
      // Choose variable occurring in the smallest $l(\varphi_i, \varphi_j)$
    if $|var_{poss}| > 1$ **then**
      $var_{poss} := var_{poss}[0]$ // Choose arbitrarily
    $Ordering := Ordering + var_{poss}$ // Append variable to *Ordering*
    $Var_{block} := Var_{block} \setminus \{var_{poss}\}$ // Remove variable from Var_{block}
    Remove var_{poss} from l
return *Ordering*

The main goal of the algorithm is to increase the chance to split the formula φ into two or more independent subformulas which is equivalent to the fact that the graph G contains two or more connected components. We do this by removing as many edges as possible with the least amount of variables assigned as possible. To achieve this, the algorithm performs the following steps:

1. The algorithm chooses variables in the order of the quantifier blocks. It selects variables in the first quantifier block until no more variables are left in it. Then, it moves on to select variables in the second quantifier block until no more variables are left in it, and so on.

2. In each quantifier block, the algorithm selects the variable that occurs the most in the edge labeling function l. It then removes this variable from the maximum number of edges in l.

3. If there is a tie between two or more variables, the algorithm chooses the variable that occurs in the smallest set of variables of the edge labeling function l. By doing this, the algorithm prioritizes variables that might remove edges in G, which generally increases the chance of achieving that G contains two or more connected components.

4. If there is still a tie between two or more variables, we resolve it arbitrarily.

5. The resulting variable is appended to the variable ordering. This variable is then removed from the set of variables of the current quantifier block. The variable must then also be removed from every set of variables of the edge labeling function l too.

Chapter 6

Prenex Normal Form

An input formula φ, defined in Definition 2.2.2, may be of an arbitrary structure. In particular, the given formula may not be in prenex normal form, as defined in Definition 2.2.3. This is problematic, as the input formula for both the decision problem meaning Algorithm 1 or Algorithm 13 and the quantifier elimination problem is expected to be in prenex normal form. To solve this problem, we need to transform any given formula into its semantically equivalent prenex normal form. Without loss of generality, it can be shown that for every first-order formula there exists an equivalent formula in prenex normal form [Hin18]. In the following, we will describe a practical approach for the transformation of an arbitrary first-order formula into its respective prenex normal form.

Algorithm 19: `to_pnf`(φ)

Input: A formula φ
Output: Tuple (q, f) where q is a list of quantifier blocks and f is a quantifier-free formula

if φ *is a constraint* **then** `// Type Atom`
  return $([], \varphi)$
if φ *is a negation* **then** `// Type Negation`
  $(q, f) :=$ `to_pnf`(φ)
  return $(\mathit{invert}(q), \neg f)$
if φ *has form* $Qx.\varphi_1$ **then** `// Type Quantifier`
  $(q, f) :=$ `to_pnf`(φ_1)
  return $(q + [Qx], f)$
φ has form $\varphi_1 \sigma \ldots \sigma \varphi_n$, $\sigma \in \{\vee, \wedge\}$ `// Type Connective`
Resolve name conflicts in $\{\varphi_1, \ldots, \varphi_n\}$
$(q, f) :=$ `to_pnf`(φ_1)
for i *in* $2, \ldots, n$ **do**
  $(q_i, f_i) :=$ `to_pnf`(φ_i)
  $q := q + q_i$
  $f := f \sigma f_i$
return (q, f)

Algorithm 19 takes a formula as input and returns a tuple (q, f) where q is a list of quantifier blocks and f is a quantifier-free formula. When calling Algorithm 19 with an arbitrary formula φ, the resulting tuple (q, f) represents exactly an equivalent formula in prenex normal form. The element q of the tuple (q, f) is a list of quantifier blocks, as defined in Definition 5.0.1, which represents the prefix of the formula in prenex normal form. The element f of the tuple (q, f) is a quantifier-free formula, which represents the matrix of the formula in prenex normal form. Algorithm 19 is a recursive algorithm that breaks down the formula into all possible subformulas and then constructs an equivalent formula in prenex normal form step by step. In the following, we are going to describe how, for each type of a formula, we calculate the respective tuple (q, f), based on the tuples $(q_1, f_1), \ldots, (q_n, f_n)$ we obtain from processing the subformulas of the given formula. Each type of a formula is handled differently.

Atom (Line 1): The input formula φ is a constraint and is thus an atom. This formula has no further subformulas. Per construction, a constraint is quantifier-free, and thus, the resulting tuple (q, f) is $([], \varphi)$.

Negation (Line 3): The input formula is of the form $\varphi := \neg\varphi_1$. In this case, the given formula has one child, the formula φ_1. The result of recursively calling $\texttt{to\_pnf}(\varphi_1)$ in stored in the tuple (q_{child}, f_{child}). The resulting tuple (q, f) for the original formula is then $(\texttt{invert}(q_{child}), \neg f_{child})$. The function `invert` is used to swap each quantifier used in q_{child} with its respective counterpart, i.e. $\forall$ is swapped with $\exists$ and vice versa. This corresponds to pulling the negation symbol from in front of the quantifiers inside.

Quantifier (Line 6): The input formula is of the form $\varphi := QB.\varphi_1$ where B is a set of variables and $Q \in \{\forall, \exists\}$ is the respective quantifier. Thus QB is a quantifier block as defined in Definition 5.0.1. This type of formula has exactly one child, the formula φ_1. The result of recursively calling $\texttt{to\_pnf}(\varphi_1)$ is stored in the tuple (q_{child}, f_{child}). The resulting tuple (q, f) for the original formula is then $(QB + q_{child}, f_{child})$, where the operator $+$ is used to denote the insertion of the quantifier block QB at the beginning of the list of quantifier blocks q_{child}.

Connective (Line 9): The input formula has the form $\varphi := \varphi_1\sigma \ldots \sigma\varphi_n$. This formula has at least two subformulas, namely $\varphi_1, \ldots, \varphi_n$. Here, we first have to resolve name conflicts in the subformulas. This is done by renaming the variables in the quantifier blocks of each subformula and in the respective formulas such that no quantified variable with a given name occurs in more than one resulting tuple. This ensures that the variables used in the quantifier blocks of the tuples are unique, and thus, when combining the quantifier blocks, no name conflicts occur. We then iterate over each subformula φ_i and recursively call $\texttt{to\_pnf}(\varphi_i) := (q_i, f_i)$ We combine the resulting lists of quantifier blocks $q_1, \ldots, q_n$ into a single list of quantifier blocks q. Similarly, we combine the quantifier-free formulas $f_1, \ldots, f_n$ into a single quantifier-free formula f using the given connective σ. Note that the order of the quantifier-free formulas is of importance, as the given connective is not necessarily commutative. The tuple (q, f) is then returned.

An example of the conversion of a first-order formula into prenex normal form is shown in Example 6.0.1.

Example 6.0.1 (Prenex normal form conversion)**.**
Let $\varphi := \neg(\forall x_1.\exists x_2.(\varphi_1(x_1, x_2) \vee (\exists x_3.(\varphi_2(x_1, x_2, x_3)))))$, where $\varphi_1(x_1, x_2)$ and $\varphi_2(x_1, x_2, x_3)$ are arbitrary constraints.

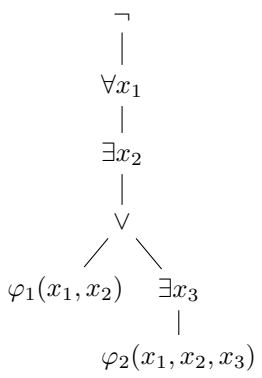


Figure 6.1: Structure of the formula φ

The structure of φ is presented in Figure 6.1. We obtain the following results when processing formula φ as input in Algorithm 19. We will present the results obtained from calling `to_pnf` for each subformula of φ.

- `to_pnf`$(\varphi_2(x_1, x_2, x_3)) \rightarrow ([], \varphi_2(x_1, x_2, x_3))$, as the given input formula is a constraint and thus an atom.

- `to_pnf`$(\exists x_3.\varphi_2(x_1, x_2, x_3))$. The given formula has one subformula, which has been processed above. We add the given quantifier block of the formula to the beginning of the list of quantifier blocks of the result of processing the subformula φ_1. This results in the tuple $([\exists x_3], \varphi_2(x_1, x_2, x_3))$.

- `to_pnf`$(\varphi_1(x_1, x_2)) \rightarrow ([], \varphi_1(x_1, x_2))$, as the given input formula is a constraint and thus an atom.

- `to_pnf`$((\varphi_1(x_1, x_2) \vee (\exists x_3.(\varphi_2(x_1, x_2, x_3)))))$. This formula has two subformulas with the results $([\exists x_3], \varphi_2(x_1, x_2, x_3))$ and $([], \varphi_1(x_1, x_2))$ as presented above. The input formula is of the connective type and thus we must first resolve the name conflicts in the list of subformula results. As the variables used in the quantifier blocks of the subformula results are pairwise disjoint, no name conflicts occur. Thus, we append the quantifier blocks of the subformula results to each other, resulting in the list of quantifier blocks $[\exists x_3]$. We then combine the quantifier-free formulas of the child results with the given boolean connective, resulting in the quantifier-free formula $\varphi_1(x_1, x_2) \vee \varphi_2(x_1, x_2, x_3)$. This results in the tuple $([\exists x_3], \varphi_1(x_1, x_2) \vee \varphi_2(x_1, x_2, x_3))$.

- `to_pnf`$(\exists x_2.(\varphi_1(x_1, x_2) \vee (\exists x_3.(\varphi_2(x_1, x_2, x_3)))))$. This formula has exactly one subformula, that has the result shown above. This formula is of the type quantifier and thus we add the outmost quantifier of the original formula to the beginning of the list of quantifier blocks of the subformula result. This results in the tuple $([\exists \{x_2, x_3\}], \varphi_1(x_1, x_2) \vee \varphi_2(x_1, x_2, x_3))$.

- `to_pnf`$(\forall x_1.\exists x_2.(\varphi_1(x_1, x_2) \vee (\exists x_3.(\varphi_2(x_1, x_2, x_3)))))$. This formula has exactly one subformula, that has the result shown above. As this formula has the type quantifier, we add the outmost quantifier of the original formula to the beginning of the list of quantifier blocks of the subformula result. This results in the tuple $([\forall\{x_1\}, \exists\{x_2, x_3\}], \varphi_1(x_1, x_2) \vee \varphi_2(x_1, x_2, x_3))$.
- `to_pnf`$(\neg(\forall x_1.\exists x_2.(\varphi_1(x_1, x_2) \vee (\exists x_3.(\varphi_2(x_1, x_2, x_3))))))$. This formula has one subformula, that has the result shown above. As this formula has the type negation, we invert the quantifier blocks of the subformula result, resulting in the list of quantifier blocks $[\exists\{x_1\}, \forall\{x_2, \exists x_3\}]$. We further negate the quantifier-free formula of the child result. This results in the tuple $([\exists\{x_1\}, \forall\{x_2, x_3\}], \neg(\varphi_1(x_1, x_2) \vee \varphi_2(x_1, x_2, x_3)))$. This tuple represents a formula in a prenex normal form equivalent to the formula φ.

Chapter 7

Discussion

In this chapter, we will discuss the implementation of the presented algorithms and evaluate their respective performance for solving quantifier-free non-linear real arithmetic formulas. This includes both the decision problem, where we are interested in the satisfiability of a given formula, as defined in Definition 2.2.7 and the quantifier-elimination problem, as defined in Definition 2.2.6. To evaluate the performance of the algorithms, we are going to compare them to state-of-the-art solver implementations. Namely, for the decision problem, we will compare our implementation to the SMT solver Z3 [DMB08] and the SMT solver CVC5 [BBB+22]. For the quantifier-elimination problem, we will compare our implementation to the QEPCAD B [Bro03] implementation of the CAD algorithm and Reduce/Redlog [DS97]. In the following, we will denote the algorithms presented in Chapter 3 as the *No Split* variant of the CAlC algorithm for quantifiers. Likewise, we are going to denote the algorithms presented in Chapter 4 as the *Split* variant of the algorithm.

7.1 Implementation

Satisfiability-Modulo-Theories Real Arithmetic Toolbox(SMT-RAT) [CKJ+15] is an open-source toolbox for strategic SMT solving written in C++. The Theory of Hybrid Systems research group maintains the solver at the RWTH Aachen University. The program has a modular structure and is designed to be easily extensible. In particular, it is possible to extend SMT-RAT to support the decision problem for non-linear real arithmetic and the quantifier-elimination problem. Over the course of this thesis, all presented algorithms have been implemented in SMT-RAT. We have based our implementation on the proof system as presented in [NÁS+22]. As a consequence, we have extended SMT-RAT by the following two modules:

- `CoveringModuleDecision`: This module implements all algorithms concerning the decision problem for non-linear real arithmetic formulas. This includes the conversion of any given input formula into a formula in prenex normal form, as presented in Chapter 6. Furthermore, both the *No Split* variant of the CAlC algorithm as presented in Chapter 3 and the *Split* variant of the CAlC algorithm as presented in Chapter 4 are included. This module contains a settings file, that allows for multiple specifications of parameters for the different algorithms. These settings include the following possible parameters:

  - The variable ordering heuristic to be used. These include the variable ordering heuristics presented in Chapter 5. The resulting variable ordering is always admissible, as defined in Definition 5.0.2.
  - A Boolean flag to choose the variant of the algorithm. This is done such that when the flag is enabled the *Split* variant of the algorithm is used. If the flag is not enabled the *No Split* variant is used.
  - A Boolean flag to set the *early_return*. This corresponds to the flag with the same name as used in the algorithms presented in Chapter 4. It only takes effect if the *Split* variant is chosen.

- `CoveringModuleQE`: This module implements all algorithms concerning the quantifier elimination problem for non-linear real arithmetic formulas. This includes Algorithm 9 and the creation of the output formula using Algorithm 10. This module contains a settings file with the same contents as in the `CoveringModuleDecision`. In Algorithm 9, either Algorithm 2 or Algorithm 14 is called depending on which variant of the algorithm is chosen in the settings file. It also includes simple formula simplifications for the generated output formula.

SMT-RAT uses *Computer ARithmetic Library*(CArL) [NÁ23], an open-source C++ library for arithmetic computations and logic, as a dependency. CArL is maintained by the Theory of Hybrid Systems research group at the RWTH Aachen University. CArL implements many sub-algorithms used in the presented algorithms, such as real root isolation, the computation of resultants and discriminants and more. Further dependencies of SMT-RAT and CArL are *Boost* [Sch11], *Libpoly* [JD17], and *CoCoALib* [AB10]. SMT-RAT also provides other tools for debugging, preprocessing and benchmarking. *Benchmax* is a tool for automated benchmarking and gathering statistics, which are defined by the used module. It also allows specifying a time and memory limit for each solver execution. Further, we extended the previously implemented parser in SMT-RAT to support the *NRA* logic. I.e., we extended the parser to support the `assert` command for non-linear real arithmetic formulas by adding the corresponding parser rules for existentially and universally quantified variables as defined in the SMT-LIB standard [BST+10]. Another addition to the parser is that fractions, with a polynomial as the denominator, may occur in the input formula. This is not supported by SMT-RAT or CArL but is allowed in the SMT-LIB standard [Ini23b]. In Section 7.2, we will discuss how we deal with this problem.

7.2 Dealing with Polynomial Denominators

As described, the SMT-LIB standard for the NRA logic allows for fractions with a polynomial as the denominator. In the theory of the reals, as defined by *SMT-LIB*, the division operator $\div$ is allowed to be used in the input formula [Ini23b]. The division operator $\div$ is defined as a function that coincides with the natural division function for all inputs x and y where y is non-zero. However, since in *SMT-LIB Logic*, all function symbols are interpreted as total functions, terms of the form $(\frac{t}{0})$ are allowed and meaningful in every instance. The declaration of such a division operator imposes no constraints on their value. In particular, this means that there is a model in the reals that satisfies the formula $(v = \frac{t}{0})$. Thus, constraints that allow the second argument of the division operator to be 0 can still be satisfiable when

there are interpretations of the functions at 0 that satisfy the constraints. This fact poses a problem for the algorithms presented in Chapter 3 and Chapter 4 since they rely on the fact that the input formula does not contain any division by a polynomial. This further poses a technical problem in the implementations in both *SMT-RAT* and *CArL*. To address this problem, we have implemented a preprocessing step that eliminates all divisions by polynomials in the input formula. This proprocessing step is taken from the preprocessing done in the NRA solver of Z3 [DMB08]. This is done during the parsing of an input formula in SMT-RAT.

- For each $\frac{p}{q}$, where q is a polynomial, we introduce a fresh variable div_{pq} and replace every occurance of $\frac{p}{q}$ with div_{pq}. Further, we add the constraint $q \neq 0 \Rightarrow div_{pq} \cdot q = p$ to the input formula.

- For each $\frac{p}{q}, \frac{p'}{q'}$ where q, q' are polynomials, we add the constraint $(p = p' \wedge q = q') \Rightarrow div_{pq} = div_{p'q'}$.

This preprocessing step ensures that the input formula does not contain any divisions by polynomials at the cost of introducing auxiliary variables and constraints. Further, this transformation of the input formula does not change the semantics of the formula. This is due to how the division operator is defined in *SMT-LIB* [Ini23b]. That is, if the division is non-zero, the semantics of the division operator coincide with the semantics of the actual division function. This preprocessing step is implemented in *SMT-RAT* and is used in all benchmarks when necessary.

7.3 The Decision Problem

In this section, we will discuss and evaluate the performance of the presented algorithms for the decision problem. In particular, we are going to evaluate the performance of the algorithms presented in Chapter 3 and Chapter 4. As above, we are going to denote the use of the algorithms presented in Chapter 3 as *No Split* and the use of the algorithms presented in Chapter 4 as *Split*. These algorithms have been implemented in SMT-RAT. As noted before, the input formula may be of an arbitrary structure and, in particular, may not be in prenex normal form. To transform any given parsed formula φ into an equivalent formula in prenex normal form when needed, we have implemented the algorithm described in Chapter 6 in SMT-RAT. The output of this algorithm is a tuple $((Q_1 x_1, \ldots, Q_n x_n), \bar{\varphi})$, with $Q_i \in \{\forall, \exists\}$ where $Q_1, \ldots, Q_n$ is the prefix and $\bar{\varphi}$ is a quantifier-free formula. It holds that $\varphi \equiv (Q_1 x_1, \ldots, Q_n x_n)\bar{\varphi}$. For both the *Split* and *No Split* approach of the cylindrical algebraic coverings methods of quantifiers, we are going to evaluate the performance of the different variable ordering heuristics presented in Chapter 5. In particular, we are interested in the runtime of the algorithms and the amount of benchmarks that the algorithms have solved. To further compare the performance of our implementation, we use the SMT solvers Z3 [DMB08] and CVC5 [BBB+22]. We have used the CVC5 version 1.0.8 and the Z3 version 4.12.2. As the basis of our evaluation, we are using the *SMT-LIB benchmark library* [Ini23a] for the logic NRA. This benchmark library provides 3819 different benchmark files. We will use a time limit of 60 seconds and a memory limit of 4 GB for each benchmark on an Intel Xeon Platinum 8160 CPU with 2.1 GHz and 48 threads. To evaluate the implementation of the algorithms presented here, we further introduce another variable ordering. The *Lexicographic Variable Ordering* sorts

	No Split			Split				Extra	
	Brown	Tri.	Lex.	Brown	Tri.	Lex.	E.S.	Z3	CVC5
SAT	3	3	2	3	3	2	3	3	3
UNSAT	3784	3793	3750	3784	3793	3750	3770	3806	3801
Unknown	30	21	63	30	21	63	44	0	0
Timeout	2	2	4	2	2	2	4	10	15
Memout	0	0	0	0	0	0	0	0	0
Avg[s]	0.023	0.023	0.023	0.022	0.022	0.022	0.021	0.033	0.029

Table 7.1: We divide the solvers into 3 categories. The first category *No Split* includes the algorithm presented in Chapter 3 i.e., the covering algorithm without a formula split in any case. The second category *Split* includes the algorithm presented in Chapter 4 i.e., the covering algorithm that includes a formula split. This category consists of 4 solvers, one for each variable ordering and one for the earliest split heuristic as presented in Algorithm 18. We further divide these two categories into the presented variable ordering heuristics. Note that we abbreviate the variable orderings as follows: *Brown* for Brown's variable ordering, *Tri.* for the triangular variable ordering and *Lex.* for the lexicographic variable ordering. Further, *E.S.* stands for the earliest split heuristic. The last category *Extra* consists of the solvers Z3 and CVC5. For each of the given solvers, we present the number of benchmark files that have returned the given result. The results are divided into the categories `SAT`, `UNSAT`, `Unknown` and `Timeout`. Further, in `Avg[s]`, the average runtime of the given solver for a solved benchmark is given in seconds.

the variables in the respective quantifier blocks lexicographically. This ordering has no practical use but is used to evaluate the impact of the variable ordering heuristics presented in Chapter 5.

In Table 7.1, we present the results of the solvers. It can be seen that both the *Split* and *No Split* implementations in SMT-RAT using the different variable ordering heuristics perform very similarly. None of the presented solvers reach the defined memory limit of 4 GB. The most significant difference in the solvers implemented in SMT-RAT can be seen in the number of benchmarks that have returned `Unknown`. These results can happen due to the use of the McCallum projection operator in the implementation of SMT-RAT [McC98]. The McCallum projection operator might fail on specific inputs, which results in the solver returning `Unknown`. This is dependent on the used variable ordering for the given input. For further detail, we refer to [McC98]. Further, we claim that the set of benchmarks does not allow for a sensible evaluation of the performance of the different variable ordering heuristics. To underline this, we also present the average time to solve a given benchmark for each solver in Table 7.1. The average runtime of any solver presented and evaluated is far below 0.1 seconds, which leaves very little information about the performance of the different solvers.

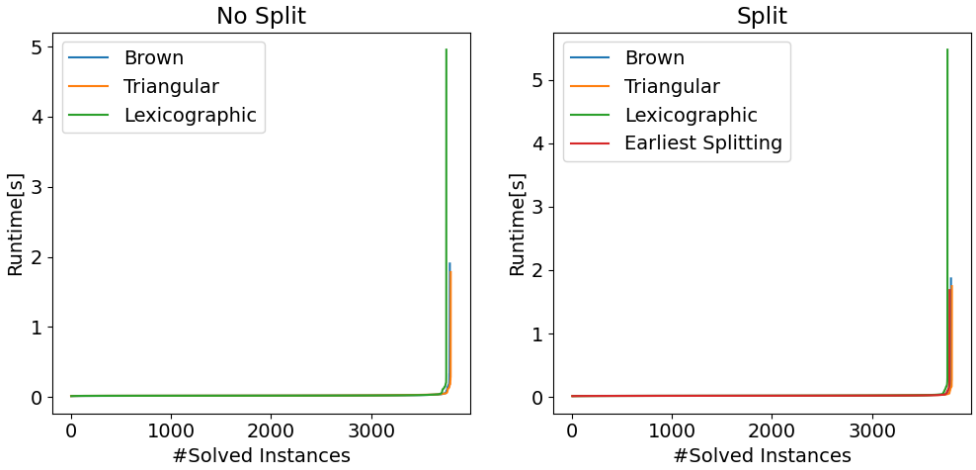


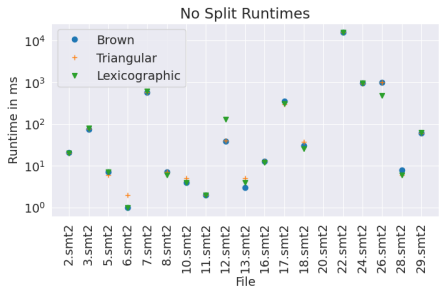
Figure 7.1: The performance profile of the solvers. The y-axis shows the runtime of the given solver, and the x-axis shows the amount of benchmarks solved by the solvers. The solvers are divided into two subplots *No Split* and *Split*. Further, each variable ordering heuristic is represented by a different color.

This fact is further illustrated in Figure 7.1. There, we visually present the performance of the solvers. The y-axis shows the runtime of the given solver, and the x-axis shows the amount of benchmarks solved by the solvers. The vast majority of benchmarks are solved in less than 0.1 seconds. This fact remains for any variable ordering heuristic implemented in SMT-RAT for both algorithms with and without formula split. When creating a formula object in SMT-RAT, some trivial simplifications are performed. This includes simplifications of the form $2 = 0 \equiv \texttt{False}$ and $\varphi(x) \vee \texttt{True} \equiv \texttt{True}$. Using these trivial simplifications, which are done implicitly when parsing any formula, 2241 formulas out of the 3819 input formulas of the used benchmarks are simplified to be trivially satisfiable or unsatisfiable. That is, satisfiability could be decided without any further computation necessary other than the parsing of the input formula. While we argue that we can not evaluate the performance of the different solvers implemented in SMT-RAT, we can still compare the performance of the solvers implemented in SMT-RAT to the performance of the solvers Z3 and CVC5 for the given benchmarks that are not trivial. For this comparison, it is sensible to use the variable ordering heuristic that produced the least amount of `Unknown` results. That is the triangular variable ordering heuristic, as defined in Definition 5.1.1 for both of the variants of the algorithm, with and without formula split. These solvers can solve 2 benchmarks, where the Z3 solver reached the time limit of 60 seconds and 12 benchmarks, where the CVC5 solver reached the time limit of 60 seconds. On the other hand, both Z3 and CVC5 cannot solve any of the benchmarks where the solvers implemented in SMT-RAT reach the time limit of 60 seconds.

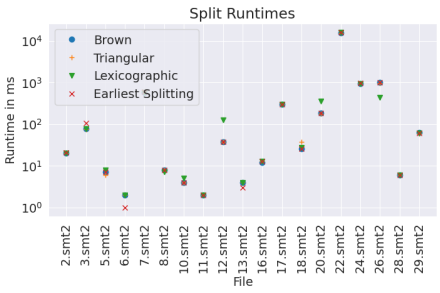
7.4 The Quantifier Elimination Problem

In the following section, we will discuss the implementation of the cylindrical algebraic coverings method for quantifiers to solve the quantifier elimination problem. Further, we will compare the performance of the presented algorithms for the quantifier elimination problem to the performance of *QEPCAD B* [Bro03] and *Reduce/Redlog* [DS97]. We have used QEPCAD B version 1.69 and Reduce rev. 6339 with the shipped version of the Redlog package. In the evaluation of the decision problem, we focus on the performance regarding runtime and memory usage. To evaluate the solvers for the quantifier elimination problem, we will also evaluate the performance of the given algorithms regarding the produced output formula. In particular, we will evaluate the size of the output formula in terms of the amount of used atoms. As already described, we have implemented Algorithm 9 and the creation of the output formula using Algorithm 10 in SMT-RAT. When in Algorithm 9 the respective next variable in the given variable ordering is quantified, we can call either Algorithm 2 or Algorithm 14 in a recursive manner. This means we can use the algorithms presented in Chapter 4 and the algorithms presented in Chapter 3. Similarly to the evaluation of the decision problem, we are going to denote the use of the algorithms presented in Chapter 3 as *No Split* and the use of the algorithms presented in Chapter 4 as *Split*. As described in Chapter 5, different variable ordering heuristics can change the performance of the used algorithms. In particular, the variable ordering heuristic might also influence the created output formula. Although the used variable ordering heuristic may influence the created output formula, we will fix the used variable order heuristic to be Brown's variable ordering, as defined in Definition 5.1.2. This variable ordering heuristic is used for both the *No Split* and the *Split* variant of the algorithm. We do this to simplify the evaluation of the different algorithms since the different variable ordering heuristics are not the focus of this thesis. QEPCAD B, Reduce/Redlog and SMT-RAT require vastly different input formats. SMT-RAT takes a file in SMT-LIB format as input. When in the given file, the *(eliminate-quantifiers ...)* or the *(apply qe)* instruction is used, SMT-RAT will eliminate all specified quantified variables in the given formula. This contrasts QEPCAD B and Reduce/Redlog, which are accessed via a command line interface and define their respective input formats. We have implemented a script to simplify the evaluation of the different algorithms, which is available at following repository: [Kro23]. This script enables us to use the same input file on all different programs. When specified, a given input file in SMT-LIB format, suitable as an input for the quantifier elimination problem in SMT-RAT, is parsed and transformed to be a suitable input for QEPCAD B and Reduce/Redlog. Notably, this includes the transformation of the input formula to be in prenex normal form for both the transformation of the inputs for QEPCAD B and Reduce/Redlog. The input file for SMT-RAT is not transformed in any way. Further, the script calls the different programs and inputs the transformed input file. It also enables setting a time limit for each program. Then, the output of the different programs is parsed and presented. When specified, the respective output formula of the programs is also stored in a file for further evaluation. For further details, we refer to the documentation, which is included in the repository of the script. All statistics presented in the following section are gathered using this script. All programs are run on an AMD Ryzen 7 5800X CPU with a time limit of 60 seconds and a memory limit of 16 GB. As a basis for the evaluation of the different algorithms, we are using exemplary quantifed formulas collected by John Wilson [Wil13]. The files themselves

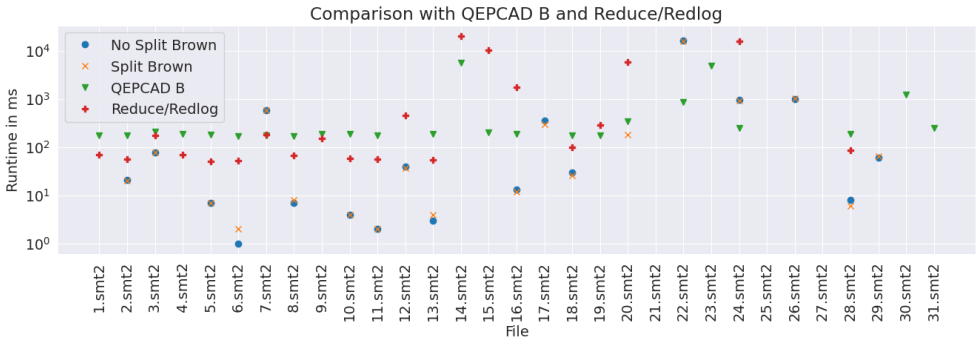
are available here[1]. There are 30 different files, each containing a single quantified formula. For all given files, the memory limit of 16 GB is not reached by any of the given programs. The output formula produced by SMT-RAT and QEPCAD B is an ETF, which is an extension to Tarski Formulas, which are Boolean combinations of polynomial equalities and inequalities [Bro99], as defined in Definition 2.3.4. The extension is that in an ETF indexed root expressions as atoms are allowed, as defined in Definition 2.3.3. In any case, a formula in ETF can be transformed into a Tarski Formula [Bro99]. The output formula produced by Reduce/Redlog is already a Tarski Formula [DS97].



(a) Runtime of the different variable ordering heuristics for the *No Split* variant of the algorithm.



(b) Runtime of the different variable ordering heuristics for the *Split* variant of the algorithm.

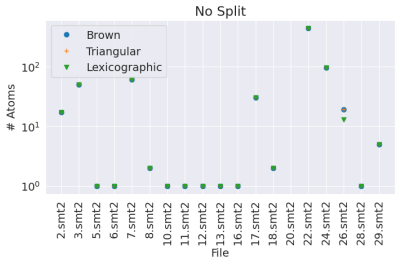


(c) Comparison of the Brown variable ordering heuristic for the *No Split* and *Split* variant of the algorithm, together with the QEPCAD B and Reduce/Redlog implementation.

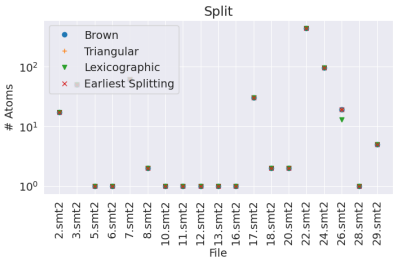
In Figures 7.2a and 7.2b, we present the runtime of the different variable ordering heuristics for the *No Split* and *Split* variant of the algorithm. On the x-axis, we present the different input files; on the y-axis, we present the runtime of the different variable ordering heuristics in milliseconds. Note that the y-axis is logarithmic. Further, we have chosen to present a subset of the benchmark files. This subset of the benchmarks is chosen such that the McCallum projection operator used in SMT-RAT does not fail on any of the given benchmarks. Therefore, when a given result is not shown in these plots, the used solver with the respective variable ordering heuristic has reached the time limit of 60 seconds. The runtime of the different variable ordering heuristics is very similar. This holds for both the *No Split* and the *Split* variant of the algorithm.

[1] `https://git.rwth-aachen.de/ths/smt/benchmarks`

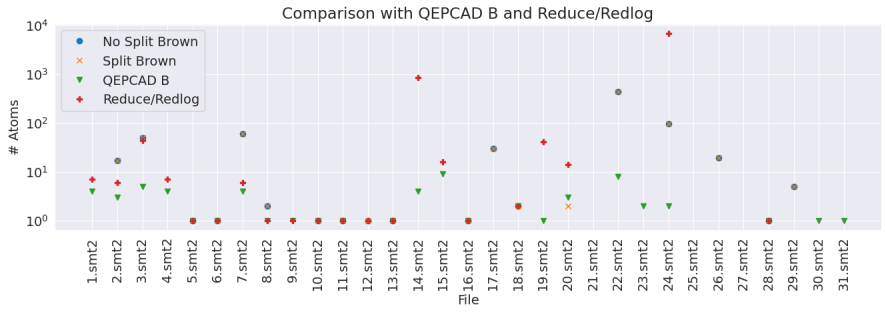
Further, it can be seen that the runtime of the *No Split* variant and the *Split* variant of the algorithm is very similar when the same variable ordering heuristic is used. This holds for all files except the file *20.smt2*. For this file, we can observe that the *No Split* variant of the algorithm reached the timeout of 60 seconds for all used variable ordering heuristics, while the *Split* variant of the algorithm did not reach the timeout for any of the used variable ordering heuristics. In Figure 7.2c, we present the runtime of the Brown variable ordering heuristic for the *No Split* and *Split* variant of the algorithm, together with the QEPCAD B and Reduce/Redlog implementation. Further, we present all benchmark files. We can observe that for select files, such as *5.smt2, 6.smt2* or *28.smt2*, our implementation in SMT-RAT outperforms the QEPCAD B and Reduce/Redlog implementation. We can even observe that for the files *26.smt2* and *29.smt2*, our implementation can solve the given benchmark while the QEPCAD B and Reduce/Redlog reach the timeout of 60 seconds. On the other hand, we can observe that QEPCAD B and Reduce/Redlog can solve more files in total in the given time and memory limit. This includes the files where the McCallum projection operator used in SMT-RAT fails.



(a) Amount of atoms in the produced output formula of the different variable ordering heuristics for the *No Split* variant of the algorithm.



(b) Amount of atoms in the produced output formula of the different variable ordering heuristics for the *Split* variant of the algorithm.



(c) Comparison of the Brown variable ordering heuristic for the *No Split* and *Split* variant of the algorithm, together with the QEPCAD B and Reduce/Redlog implementation.

In Figures 7.3a and 7.3b, we present the number of atoms in the produced output formula of the different variable ordering heuristics for the *No Split* and *Split* variant of the algorithm. On the x-axis, we present the different input files; on the y-axis, we present the number of atoms in the produced output formula of the different variable ordering heuristics. Analogously to Section 7.4, the y-axis is logarithmic, and we have chosen to present a subset of the benchmark files in which the McCallum operator

implemented in SMT-RAT does not fail. Interestingly, the used variable ordering heuristic generally does not affect the number of atoms in the produced output formula, except in one case, namely the file *26.smt2*. Here, for both the *No Split* and the *Split* variant of the algorithm, the lexicographic variable ordering heuristic produces an output formula with fewer atoms than the other variable ordering heuristics. In Figure 7.3c, we present the number of atoms in the produced output formula of the Brown variable ordering heuristic for the *No Split* and *Split* variant of the algorithm, together with the QEPCAD B and Reduce/Redlog implementation. Further, we present all benchmark files. Firstly, we observe that Reduce/Redlog produces large output formulas in two of the given benchmark files, while the other solvers can produce a significantly smaller output formula. Here, we note again that SMT-RAT and QEPCAD B produce an ETF, while Reduce/Redlog produces a Tarski Formula. We further observe that the number of atoms in the output formula produced by the QEPCAD B implementation is generally smaller than that produced by the SMT-RAT implementation. Based on the presented data, we conclude that QEPCAD B is better at producing smaller output formulas than the implementation of the presented algorithms in SMT-RAT.

Chapter 8

Conclusion

In this thesis, we have introduced the quantifier elimination problem and the decision problem for non-linear real arithmetic, a first-order theory of multivariate polynomial constraints over real numbers. The quantifier elimination problem involves finding an equivalent quantifier-free formula for a given formula. The decision problem only requires determining whether a given formula is satisfiable. We have presented the Cylindrical Algebraic Covering algorithm for quantifiers, which is a conflict-driven approach for deciding the satisfiability of a formula. This method can solve both the decision problem and the quantifier elimination problem. Further, we have presented a novel adaptation of this algorithm, which allows for the split of the input formula into two or more independent subformulas. These subformulas can then be solved independently, giving the result for the original formula. We have implemented the original algorithm and the novel adaptation in the SMT solver SMT-RAT. Further, we have introduced different variable ordering heuristics, two of which have been shown to be effective in literature. Additionally, we have presented a novel variable ordering heuristic, which aims to achieve the split of the input formula into two or more independent subformulas with the least amount of assigned variables. We have evaluated the performance of the original algorithm and the novel adaptation for both the decision problem and the quantifier elimination problem. Further, we have compared these results with other state-of-the-art solvers, namely Z3 and CVC5 for the decision problem and QEPCAD B and Reduce/Redlog for the quantifier elimination problem. We have shown that our implementations are competitive with the other solvers for the decision problem for the used NRA benchmarks, consisting of 3819 files, and even outperform Z3 and CVC5 on specific files. We can further observe that our implementation of the adaptation of the formula split achieves neither a gain nor a loss in performance for the used benchmarks. That may be due to the fact that the used benchmarks are not complicated enough to benefit from the input split of the formula into independent subformulas. The implementations in SMT-RAT can solve 2 benchmarks the Z3 solver could not solve and 12 benchmarks the CVC5 solver could not solve within the given resource limits. On the other hand, the other solvers cannot solve any benchmarks for which the SMT-RAT implementations reach the resource limits. The biggest downfall of our implementation is the use of the McCallum projection operator, due to which an inconclusive result for up to 63 benchmarks is returned. In our implementation for the quantifier elimination problem, we can reuse the same procedures as in the decision problem, which further allows the novel

adaptation of the split of the formula into two or more independent subformulas when suitable. When evaluating the performance of the presented algorithms for the quantifier elimination problem, we are interested in the resource usage of the solver and also the quality of the produced formula, for which we count the number of atoms. The presented algorithms in SMT-RAT have shown competitive performance compared to the QEPCAD B and Reduce/Redlog implementations in solving benchmark files within the given time and memory limit. We observe that neither the choice of variable ordering heuristic nor the use of the split of the formula into independent subformulas majorly influences the performance of the presented algorithms for the used test files. Further, we observe that our implementations can outperform both QEPCAD B and Reduce/Redlog for specific files regarding running time. On the other hand, we can observe that QEPCAD B and Reduce/Redlog can solve more files in total in the given time and memory limit. This includes the files where the McCallum projection operator used in SMT-RAT fails. However, QEPCAD B has shown to be more efficient in producing smaller output formulas than our implementations.

8.1 Future Work

In this section, we present some ideas for future work which could improve the performance of the presented algorithms.

- **Implementing a new projection operator:** As we have seen in the evaluation of the presented algorithms, the McCallum projection operator is the biggest downfall of our implementation. Therefore, it would be interesting to implement a new projection operator that does not have the same drawbacks as the McCallum projection operator. One possible candidate for such a projection operator is the Lazard projection operator presented in [Laz94].

- **Heuristic for when to split:** In our current implementation, we split the formula whenever possible or not at all. For this, it would be interesting to develop a heuristic that decides when to split the formula and when not to split the formula. This might include that the formula is only split given a certain threshold of the number of variables in the formula or similar. This may reduce the overhead of the split of the formula into independent subformulas when there is no potential gain in performance.

- **Implementing a new variable ordering heuristic:** We have presented a variable ordering heuristic that aims to achieve the split of the formula into two or more independent subformulas with the least amount of assigned variables. However, this variable ordering heuristic may not achieve the optimal variable ordering for this goal. Therefore, it would be interesting to develop an exact algorithm that produces the optimal variable ordering for achieving the split of the input formula into two or more independent subformulas with the least amount of assigned variables possible.

- **Proof of Concept Formula Evaluation:** When checking if two subformulas are independent with respect to a variable and a partial sample point, we need to evaluate the subformulas over the sample point and calculate the variables that are present in both subformulas. In the current implementation, we evaluate the subformulas over the sample point and then calculate the intersection of

the variables without storing intermediate information. However, storing this information for later use might be sensible to check if two subformulas are independent with respect to some other variable and an extension of the sample point. Further, the implementation of this formula evaluation is completely separate from the partial formula evaluation done in the `forall` or `exists` algorithm of the CAlC algorithm and could be unified for further perfomance gain.

- **Individual Variable Ordering for each Branch:** In the current implementation, we calculate the variable ordering before the start of the CAlC algorithm. This variable ordering is then used for all branches of the CAlC algorithm and never changed. However, it has been shown that the variable ordering may be changed for each branch of the CAlC algorithm to improve performance [DSS04]. Thus, adjusting the implementation such that after each split of a given formula, a new variable ordering is calculated for each subformula, might improve the performance of the presented algorithm.

Bibliography

[AB10] John Abbott and Anna M Bigatti. Cocoalib: A c++ library for computations in commutative algebra... and beyond. In *Mathematical Software–ICMS 2010: Third International Congress on Mathematical Software, Kobe, Japan, September 13-17, 2010. Proceedings 3*, pages 73–76. Springer, 2010.

[ÁDEK21] Erika Ábrahám, James H Davenport, Matthew England, and Gereon Kremer. Deciding the consistency of non-linear real arithmetic constraints with a conflict driven search using cylindrical algebraic coverings. *Journal of Logical and Algebraic Methods in Programming*, 119:100633, 2021.

[BBB+22] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres 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*, pages 415–442. Springer, 2022.

[BDE+16] Russell Bradford, James H Davenport, Matthew England, Scott McCallum, and David Wilson. Truth table invariant cylindrical algebraic decomposition. *Journal of Symbolic Computation*, 76:1–35, 2016.

[Bro99] Christopher W Brown. *Solution formula construction for truth invariant CAD's*. University of Delaware, 1999.

[Bro03] Christopher W Brown. Qepcad b: a program for computing with semi-algebraic sets using cads. *ACM Sigsam Bulletin*, 37(4):97–108, 2003.

[BST+10] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, volume 13, page 14, 2010.

[BT18] Clark Barrett and Cesare Tinelli. *Satisfiability modulo theories*. Springer, 2018.

[CH91] George E Collins and Hoon Hong. Partial cylindrical algebraic decomposition for quantifier elimination. *Journal of Symbolic Computation*, 12(3):299–328, 1991.

[CJ12] Bob F Caviness and Jeremy R Johnson. *Quantifier elimination and cylindrical algebraic decomposition*. Springer Science & Business Media, 2012.

[CKJ+15] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. SMT-RAT: an open source C++ toolbox for strategic and parallel SMT solving. In Marijn Heule and Sean A. Weaver, editors, *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, volume 9340 of *Lecture Notes in Computer Science*, pages 360–368. Springer, 2015.

[Col76] George E Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition: a synopsis. *ACM SIGSAM Bulletin*, 10(1):10–12, 1976.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery.

[DH88] James H Davenport and Joos Heintz. Real quantifier elimination is doubly exponential. *Journal of Symbolic Computation*, 5(1-2):29–35, 1988.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[DS97] Andreas Dolzmann and Thomas Sturm. Redlog: Computer algebra meets computer logic. *Acm Sigsam Bulletin*, 31(2):2–9, 1997.

[DSS04] Andreas Dolzmann, Andreas Seidl, and Thomas Sturm. Efficient projection orders for cad. In *Proceedings of the 2004 international symposium on Symbolic and algebraic computation*, pages 111–118, 2004.

[EBDW14] Matthew England, Russell Bradford, James H Davenport, and David Wilson. Choosing a variable ordering for truth-table invariant cylindrical algebraic decomposition by incremental triangular decomposition. In *Mathematical Software–ICMS 2014: 4th International Congress, Seoul, South Korea, August 5-9, 2014. Proceedings 4*, pages 450–457. Springer, 2014.

[EF19] Matthew England and Dorian Florescu. Comparing machine learning models to choose the variable ordering for cylindrical algebraic decomposition. In *Intelligent Computer Mathematics: 12th International Conference, CICM 2019, Prague, Czech Republic, July 8–12, 2019, Proceedings 12*, pages 93–108. Springer, 2019.

[HEW+14] Zongyan Huang, Matthew England, David Wilson, James H Davenport, Lawrence C Paulson, and James Bridge. Applying machine learning to the problem of choosing a heuristic to select the variable ordering for cylindrical algebraic decomposition. In *Intelligent Computer Mathematics: International Conference, CICM 2014, Coimbra, Portugal, July 7-11, 2014. Proceedings*, pages 92–107. Springer, 2014.

[Hin18] Peter G Hinman. *Fundamentals of mathematical logic*. CRC Press, 2018.

[Hon90] Hoon Hong. An improvement of the projection operator in cylindrical algebraic decomposition. In *Proceedings of the international symposium on Symbolic and algebraic computation*, pages 261–264, 1990.

[Ini23a] The SMT-LIB Initiative. Smt-lib nra benchmarks, October 2023. `https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/NRA`.

[Ini23b] The SMT-LIB Initiative. Smt-lib nra theory, October 2023. `https://smtlib.cs.uiowa.edu/theories-Reals.shtml`.

[JA95] Clark John and Holton Derek Allan. *A first look at graph theory*. Allied Publishers, 1995.

[JD17] Dejan Jovanovic and Bruno Dutertre. Libpoly: A library for reasoning about polynomials. In *SMT*, pages 28–39, 2017.

[KN22] Gereon Kremer and Jasper Nalbach. Cylindrical algebraic coverings for quantifiers. 2022.

[Kro23] Philip Kroll. Quantifier benchmark, November 2023. `https://zenodo.org/records/10221072?token=eyJhbGciOiJIUzUxMiJ9.eyJpZCI6IjMzODIwM2FiLWUyZTAtNDkxNy04NWJhLTczZjNkZmU5ZGIwMiIsImRhdGEiOnt9LCJyYW5kb20iOiI0NGQyYTk5NDNlNTk0ZjA5ZjY0ZmMxZjg3OTk3OWJiYyJ9.f4KsnV7mTORq28C8dTRQhPdfQkceJltszX6eDgrUFQZ6jveJJJv5pC1YP_A6zrBpwovq1r-ILf2MxUuP6vby8w`.

[Laz94] Daniel Lazard. An improved projection for cylindrical algebraic decomposition. In *Algebraic geometry and its applications: collections of papers from Shreeram S. Abhyankar's 60th birthday conference*, pages 467–476. Springer, 1994.

[LXZZ21] Haokun Li, Bican Xia, Huiying Zhang, and Tao Zheng. Choosing the variable ordering for cylindrical algebraic decomposition via exploiting chordal structure. In *Proceedings of the 2021 on International Symposium on Symbolic and Algebraic Computation*, pages 281–288, 2021.

[McC98] Scott McCallum. An improved projection operation for cylindrical algebraic decomposition. In *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 242–268. Springer, 1998.

[NÁ23] Florian Corzilius Gereon Kremer Sebastian Junges Stefan Schupp Jasper Nalbach and Erika Ábrahám. Computer ARithmetic Library CArL, October 2023. `https://github.com/ths-rwth/carl`.

[NÁS+22] Jasper Nalbach, Erika Ábrahám, Philippe Specht, Christopher W Brown, James H Davenport, and Matthew England. Levelwise construction of a single cylindrical algebraic cell. *arXiv preprint arXiv:2212.09309*, 2022.

[Sch11] Boris Schling. *The Boost C++ Libraries*. XML Press, 2011.

[Sei54] Abraham Seidenberg. A new decision method for elementary algebra. *Annals of Mathematics*, pages 365–374, 1954.

[Tar51] Alfred Tarski. A decision method for elementary algebra and geometry. In *Quantifier elimination and cylindrical algebraic decomposition*, pages 24–84. Springer, 1951.

[Wil13] David Wilson. Real geometry and connectedness via triangular description: Cad example bank, April 2013. Each example is given as a Tarski formula or list of polynomials followed by a list of free variables, a list of quantified variables, the suggested variable order given from the source (if any), the minimal number of cells achieved in a full CAD (with details of how to reproduce), notes on the problem, and the source.