Theory of Hybrid Systems Informatik 2
RWTH AACHEN UNIVERSITY

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

BACHELOR OF SCIENCE THESIS

USING HORNER SCHEMES TO IMPROVE THE EFFICIENCY AND PRECISION OF INTERVAL CONSTRAINT PROPAGATION

Lukas Netz

Examiners:
Prof. Dr. Erika Ábrahám
Prof. Dr. Jürgen Giesl
Additional Advisors:
Stefan Schupp, Florian Corzilius

Aachen, 28.9.2015

Abstract

Interval Constraint Propagation (ICP) is a powerful procedure that applies contraction methods to narrow down the solution space of *quantifier-free non-linear real arithmetic* (QFNRA) problems. However, interval arithmetic suffers from the dependency problem and other effects, that cause over-approximations within basic arithmetic operations, which lower the efficiency and precision of ICP.

In this thesis we present an approach to improve the performance of an ICP module within the satisfiability modulo theories (SMT) framework SMT-RAT, a toolbox which allows to compose a SMT solver via custom modules. We improve the contraction via interval propagation and introduce a new data structure for multivariate Horner schemes. By reducing the total amount of mathematical operations and by rearranging polynomials we aim to reduce both the wrapping effect and over-approximation, which is inherent to the used interval arithmetic.

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.

Lukas Netz
Aachen, den 28. November 2015

Acknowledgements

At this point I want to express my gratitude to Prof. Dr. Ábrahám, who gave me the opportunity to write this bachelor thesis. I want to thank Stefan Schupp deeply, who could always spare some time to give me helpful advice, and who patiently answered all of my questions. I also owe thanks to Florian Corzilius, who invested a lot of time conducting benchmark tests for me and supported me during my work for the thesis. Finally I want to thank my friends and family for their support and encouragement, helping me manage even the most work intense phases of the thesis.

Contents

Chapter 1

Introduction

In the last decades the demand for algorithms to check satisfiability of systems of quantifier-free non-linear real arithmetic (QFNRA) constraints over real numbers has increased significantly. Many applications in science and engineering require fast and efficient methods for satisfiability checking of QFNRA formulas, like conjunctions of polynomials or terms. While the interval constraint propagation provides a good procedure to handle these problems, it suffers from over approximation due to the wrapping effect and other influences that decline it's performance.

Satisfiability modulo theories (SMT) is a viable approach to solve these problems. SMT-solvers are based on a combination of a highly efficient satisfiability (SAT) solvers and powerful theory solvers. QFNRA formulas are hard to solve, existing complete solvers for nonlinear formulas with polynomial functions have at least an exponential complexity [6]. Some procedures that are capable to test the satisfiability of QFNRA formulas are cylindrical algebraic decomposition, virtual substitution and Gröbner bases, although both latter procedures are incomplete, which means they cannot determine the satisfiability for every formula they get as input. Interval constraint propagation (ICP) is another incomplete method. This thesis aims to provide an improvement for ICP by providing a Horner scheme based data structure. It reduces the search space of QFNRA formulas. ICP cannot determine for every input formula its satisfiability, but if possible it will return an infeasible subset. Within the SMT solver the ICP procedure works as a preprocessor and is able to provide a consecutive solver with a smaller search space.

The Horner scheme, named after the British mathematician William George Horner [11], is a transformation for polynomials and known to provide several useful features. One of them is the reduction of arithmetic operations, that are needed to solve the polynomial. This provides a more efficient environment for computers to work with polynomials. We analyze the effects of Horner scheme based interval constraint propagation and discuss whether it is viable to transform the provided polynomials, in order to improve the overall precision and efficiency of the ICP procedure.

For this thesis we use an SMT solver, called SMT-RAT [3], that was developed at our chair. It allows to combine multiple custom solver modules to form a composed solver. We optimize the existing ICP module within SMT-RAT, which has been developed by S. Schupp [16] [9]. The implementation of the multivariate Horner scheme data structure faces some difficulties: On the one hand the data structure has to be fully integrable with the preexisting code structures, on the other hand efficient coding is required, in order to provide notable improvements within the highly optimized framework.

Our goal is to improve the contraction procedures within the ICP module by using Horner schemes in the evaluation process of polynomials, and by providing another contraction method (constraint propagation). We aim to speed up the solving process while increasing the size of the contractions. We will try to realize the Horner schemes by implementing the concepts postulated in a paper from Martine Ceberio and Vladik Kreinovich [7].

This thesis consists of six chapters. In the next Chapter after the introduction, this thesis will commence by presenting the preliminaries. We will introduce two interval arithmetics, the basics of the ICP procedure and get in detail with the different contraction methods used by the ICP module. In Chapter 3 we present the characteristics of Horner schemes. We will examine their effects on interval arithmetic, especially on precision and efficiency. We will also highlight two heuristics for the creation of multivariate Horner schemes, that were implemented within the data structure. In Chapter 4 we will present the key functions of our implementation, highlighting their basic concepts and explaining the data structure used to realize multivariate Horner schemes. In Chapter 5 we take a look at the results of our work, and will analyze the performance of the modified ICP procedure and compare it to the default setting of the framework. In the final Chapter 6 we will draw a conclusion and point out concepts for future work.

Chapter 2

Preliminaries

In this chapter we will provide a basic introduction for the mathematical operations and procedures used in this thesis. We will introduce two interval arithmetics, and focus on over-approximation caused by the dependency problem. We shortly present *satifiability modular theories* (SMT) solving and the framework SMT-RAT, followed by a presentation of *interval constraint propagation* (ICP) and its contraction procedures.

2.1 Interval Arithmetic

ICP makes use of interval arithmetic at many points within its procedure. In order to follow its principle of operation we need to introduce some basic arithmetic operations. In the following we will present two different interval arithmetics, which differ in the representation of the intervals and thus provide different approaches towards basic arithmetic operations. To differentiate the two, we refer to the interval arithmetic used in the paper from Ulrich W. Kulisch [13] as *standard interval arithmetic* and the interval arithmetic used by Siegfried M. Rump [15] as *midpoint-radius interval arithmetic*. It is necessary to introduce both interval arithmetics, because the interval arithmetic used in the paper by Martine Ceberio and Vladik Kreinovich [7] differs from the one we use in our implementation.

2.1.1 Standard Interval Arithmetic

In this section we present basic mathematical operations of interval arithmetic as implemented by S. Schupp for the ICP procedure [16]. We start by defining an interval for standard interval arithmetic:

Definition 2.1.1 (Interval). *An Interval $I \subseteq \mathbb{R}$ is a set of numbers, such that there exists $a,b \in \mathbb{R} \cup \{-\infty, +\infty\}$ such that $I = \{x \in \mathbb{R} | a \backsim_l x \backsim_u b\}$ for some $\backsim_l$, $\backsim_u \in \{<, \leq\}$.*

In the following the set of intervals, which satisfy the definition above, are referred to as $\mathbb{IR}$. We use the following notation to express including endpoints of an interval, also known as weak bounds $[a; b]$ and excluding endpoints of an interval, also known as strict bounds $(a; b)$:

$$\begin{aligned}(a;b) &= \{x \in \mathbb{R} \mid a < x < b\}\\ [a;b) &= \{x \in \mathbb{R} \mid a \leq x < b\}\\ (a;b] &= \{x \in \mathbb{R} \mid a < x \leq b\}\\ [a;b] &= \{x \in \mathbb{R} \mid a \leq x \leq b\}\end{aligned}$$

Definition 2.1.2 (Intersection). *For two intervals $I_a, I_b \in \mathbb{IR}$ we define an interval intersection as follows:*

$$I_a \cap I_b = \{x | x \in I_a \wedge x \in I_b\}$$

Note that the intersection of two intervals will always result in the maximal interval that satisfies both bound types.

Definition 2.1.3 (Union). *For two intervals $I_a, I_b \in \mathbb{IR}$ we define an interval union as follows:*

$$I_a \cup I_b = \{x | x \in I_a \vee x \in I_b\}$$

In order to describe the search space for an n-Dimensional equation we use an interval box:

Definition 2.1.4 (Interval box). *An n-dimensional vector of intervals $B = (x_1, ..., x_n)$ with $B \in \mathbb{IR}^n$ is called an interval box.*

As the ICP module only needs *addition*, *subtraction*, *multiplication* and *division*, we focus only on those operations in interval arithmetic. When defining mathematical operations for intervals, we have to consider all possible combinations of parameters, such that the resulting interval contains all results of the respective operation applied on all combinations of values taken from the input intervals:

$$I_a \odot I_b = \{a \odot b \mid a \in I_a, b \in I_b\} \tag{2.1}$$

In the following we define the operations for addition, subtraction, multiplication and division with intervals of the standard interval arithmetic.

Standard interval addition: When adding up two intervals, we have to consider both the lower and upper bound of the intervals. The bounds of the resulting interval are each the sum of the start- and endpoints of the respective input intervals.

Definition 2.1.5 (Standard interval addition). *The addition of two intervals $A = [a_1,a_2]$ and $B = [b_1,b_2]$, where $A,B \in \mathbb{IR}$, is defined as:*

$$[a_1; a_2] + [b_1; b_2] = [a_1 + b_1; a_2 + b_2]$$

A definition for standard interval addition in consideration of infinite bounds is displayed in Table 2.1.

Example 2.1.1 (Standard interval addition). *Adding $[-4; 2]$ and $[1; 4]$ yields to:*

$$[-4; 2] + [1; 4] = [-4 + 1; 2 + 4] = [-3; 6]$$

Addition	$(-\infty, b_2]$	$[b_1, b_2]$	$[b_1, +\infty)$	$(-\infty, +\infty)$
$(\infty, a_2]$	$(\infty, a_1 + b_2]$	$(-\infty, a_2 + b_2]$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1,a_2]$	$(-\infty, a_2 + b_2]$	$[a_1 + b_1, a_2 + b_2]$	$[a_1 + b_1, +\infty)$	$(-\infty, +\infty)$
$[a_1, +\infty)$	$(-\infty, +\infty)$	$[a_1 + b_1, +\infty)$	$[a_1 + b_1, +\infty)$	$(-\infty, +\infty)$
$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$

Table 2.1: Definition for standard interval addition with consideration of infinity, [13].

Standard interval subtraction: Similar to standard interval addition we use the lower and upper bounds to calculate the resulting interval.

Definition 2.1.6 (Interval subtraction). *The subtraction of two intervals $A = [a_1; a_2]$ and $B = [b_1; b_2]$, where $A,B \in \mathbb{IR}$, is defined as:*

$$[a_1; a_2] - [b_1; b_2] = [a_1 - b_2; a_2 - b_1]$$

A definition for standard interval subtraction in consideration of infinite bounds is displayed in Table 2.2.

Example 2.1.2 (Standard interval subtraction). *Subtracting $[6; 9]$ and $[2; 3]$ yields to:*

$$[6; 9] - [2; 3] = [6 - 3; 9 - 2] = [3; 7]$$

Subtraction	$(-\infty, b_2]$	$[b_1, b_2]$	$[b_1, +\infty)$	$(-\infty, +\infty)$
$(-\infty, a_2]$	$(-\infty, +\infty)$	$(-\infty, a_2 - b_1]$	$(-\infty, a_2 - b_1]$	$(-\infty, +\infty)$
$[a_1, a_2]$	$[a_1 - b_2, +\infty]$	$[a_1-_2, a_2 - b_1]$	$(-\infty, a_2 - b_1]$	$(-\infty, +\infty)$
$[a_1, +\infty)$	$[a_1 - b_2, +\infty)$	$[a_1 - b_2, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$

Table 2.2: Definition for standard interval subtraction with consideration of infinity [13].

Standard interval multiplication: Up to now we were able to use closed formulas for arithmetic operations to add up or subtract intervals, using the bounds of the input intervals. We might encounter a problem if we use that same principle in interval multiplication:

Example 2.1.3 (Standard interval multiplication)**.** *If we multiply both boundaries pairwise and both intervals start with a negative number, the resulting interval should also contain a negative number, if at least one of the intervals contains a positive number.*

$$[-4;3] \cdot [-1;4] \neq [-4 \cdot -1; 3 \cdot 4] = [4;12]$$

-2 *is a value within the first interval and* 1 *is a value within the second interval, thus* $-2 \cdot 1 = -2$ *should be contained be the resulting interval. The correct result would be:*

$$[-4;3] \cdot [-1;4] = [-4 \cdot 4; 3 \cdot 4] = [-8;12]$$

We have to guarantee that all possible results of multiplications from values within the first interval and values within the second interval are contained within the resulting interval (see Equation 2.1). Therefore we have to take a different approach in order to calculate the lower and the upper bound of the resulting interval. We define interval multiplication as follows:

Definition 2.1.7 (Standard interval multiplication)**.** *The multiplication of two intervals* $A = [a_1; a_2]$ *and* $B = [b_1; b_2]$*, where* $A,B \in \mathbb{IR}$*, is defined as:*

$$[a_1; a_2] \cdot [b_1; b_2] = [min\{a_1 \cdot b_1, a_1 \cdot b_2, a_2 \cdot b_1, a_2 \cdot b_2\}; max\{a_1 \cdot b_1, a_1 \cdot b_2, a_2 \cdot b_1, a_2 \cdot b_2\}]$$

Note that it is possible to define the operations for plus and minus the same way:

$$[a_1,a_2] + [b_1,b_2] = [min\{a_1 + b_1, a_1 + b_2, a_2 + b_1, a_2 + b_2\} \ ; \ max\{a_1 + b_1, a_1 + b_2, a_2 + b_1, a_2 + b_2\}]$$
$$[a_1,a_2] - [b_1,b_2] = [min\{a_1 - b_1, a_1 - b_2, a_2 - b_1, a_2 - b_2\} \ ; \ max\{a_1 - b_1, a_1 - b_2, a_2 - b_1, a_2 - b_2\}]$$

We use addition and subtraction as defined in Definition 2.1.5 and Definition 2.1.6, because they provide a cheaper and more efficient way to obtain the same result. A definition for standard interval multiplications in consideration of infinite bounds is displayed in Table 2.3.

Standard interval division: The division of standard intervals follows the same principle as multiplication. However, we have to consider a divisor that might contain zero:

Example 2.1.4 (Standard interval division)**.** *Divisor not containing zero:* $[4;8]$ *divided by* $[1;2]$ *yields to:*

$$[4;8] \div [1;2] = [4 \div 2; 8 \div 1] = [2;8]$$

Divisor containing zero: $[4;8]$ *divided by* $[-1;2]$ *yields to:*

$$[4;8] \div [-1;2] = (-\infty; 4 \div -1] \cup [4 \div 2; +\infty) = (-\infty; -4] \cup [2; +\infty)$$

In case the divisor contains zero, the division operator will split the interval. We refer to this as an *heteronomous split*. We are able to define interval division based on Equation 2.1 and standard interval multiplication (Definition 2.1.7):

Definition 2.1.8 (Standard interval division)**.** *The division of two intervals* $A = [a_1; a_2]$ *and* $B = [b_1; b_2]$*, where* $A,B \in \mathbb{IR}$*, is defined as:*

$$A/B := \{x \mid bx = a \wedge a \in A \wedge b \in B\}$$

A definition for interval division with a divisor not containing zero is displayed in Table 2.6, a definition for interval division with a divisor not containing zero and in consideration of infinite bounds is displayed in Table 2.5, a definition for interval division with a divisor containing zero is displayed in Table 2.4.

The arithmetic operations stated above, are presented for standard intervals with weak bounds. The arithmetic does not depend on the bound type and is therefore derivable from the standard interval arithmetic as the same mathematical principles apply for both bound types.

In the following intervals of the standard interval arithmetic are referred to simply as 'intervals'.

Multiplication	$[b_1,b_2]$ $b_2 \leq 0$	$[b_1,b_2]$ $b_1 < 0 < b_2$	$[b_1,b_2]$ $b_1 \geq 0$	[0,0]	$(-\infty, b_2]$ $b_2 \leq 0$	$(-\infty, b_2]$ $b_2 \geq 0$	$[b_1, +\infty)$ $b_1 \leq 0$	$[b_1, +\infty)$ $b_1 \geq 0$	$(-\infty, +\infty)$
$[a_1, a_2], a_2 \leq 0$	$[a_2 \cdot b_2, a_1 \cdot b_1]$	$[a_1 \cdot b_2, a_1 \cdot b_1]$	$[a_1 \cdot b_2, a_2 \cdot b_1]$	[0,0]	$[a_2 \cdot b_2, +\infty)$	$[a_1 \cdot b_2, +\infty)$	$(-\infty, a_1 \cdot b_2]$	$(-\infty, a_2 \cdot b_1]$	$(-\infty, +\infty)$
$a_1 < 0 < a_2$	$[a_2 \cdot b_1, a_1 \cdot b_1]$	$[\min\{a_1 \cdot b_2, a_2 \cdot b_1\}, \max\{a_1 \cdot b_1, a_2 \cdot b_2\}]$	$[a_1 \cdot b_2, a_2 \cdot b_2]$	[0,0]	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, a_2], a_1 \geq 0$	$[a_2 \cdot b_1, a_1 \cdot b_2]$	$[a_2 \cdot b_1, a_2 \cdot b_2]$	$[a_1 \cdot b_1, a_2 \cdot b_2]$	[0,0]	$(-\infty, a_1 \cdot b_2]$	$(-\infty, a_2 \cdot b_2]$	$[a_2 \cdot b_2, +\infty)$	$[a_1 \cdot b_1, +\infty)$	$(-\infty, +\infty)$
[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]
$(-\infty, a_2], a_2 \leq 0$	$[a_2 \cdot b_2, +\infty)$	$(-\infty, +\infty)$	$(-\infty, a_2 \cdot b_1]$	[0,0]	$[a_2 \cdot b_2, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, a_2 \cdot b_1]$	$(-\infty, +\infty)$
$(-\infty, a_2], a_2 \geq 0$	$[a_2 \cdot b_1, +\infty)$	$(-\infty, +\infty)$	$(-\infty, a_2 \cdot b_2]$	[0,0]	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, +\infty), a_1 \leq 0$	$(-\infty, a_1 \cdot a_1]$	$(-\infty, +\infty)$	$[a_1 \cdot b_2, +\infty)$	[0,0]	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, +\infty), a_1 \geq 0$	$(-\infty, a_1 \cdot b_2]$	$(-\infty, +\infty)$	$[a_1 \cdot b_1, +\infty)$	[0,0]	$(-\infty, a_1 \cdot b_2]$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$[a_1 \cdot b_1, +\infty)$	$(-\infty, +\infty)$
$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	[0,0]	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$

Table 2.3: Definition for standard interval multiplication with consideration of infinity [13].

Division $0 \in B$	$B =$ [0,0]	$[b_1,b_2]$ $b_1 < b_2 = 0$	$[b_1,b_2]$ $b_1 < 0 < b_2$	$[b_1,b_2]$ $0 = b_1 < b_2$	$(-\infty, b_2]$ $b_2 = 0$	$(-\infty, b_2]$ $b_2 > 0$	$[b_1, +\infty)$ $b_1 < 0$	$[b_1, +\infty)$ $b_1 = 0$	$(-\infty, +\infty)$
$[a_1,a_2], a_2 < 0$	$\varnothing$	$[a_2/b_1, +\infty)$	$(-\infty, a_2/b_2] \cup [a_2/b_1, +\infty)$	$(-\infty, a_2/b_2]$	$[0, +\infty)$	$(-\infty, a_2/b_2] \cup [0, +\infty)$	$(-\infty,0] \cup [a_2/b_1, +\infty)$	$(-\infty,0]$	$(-\infty, +\infty)$
$[a_1,a_2], a_1 \leq 0 \leq a_2$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1,a_2], a_1 > 0$	$\varnothing$	$(-\infty, a_1/b_1]$	$[a_1/b_2, +\infty) \cup [a_1/b_2, +\infty)$	$(-\infty,0]$	$(-\infty,0]$	$(-\infty, a_1/b_1] \cup [a_1/b_2, +\infty)$	$(-\infty, a_1/b_1]$ $[0, -\infty)$	$[0, +\infty)$	$(-\infty, +\infty)$
$(-\infty,a_2], a_2 < 0$	$\varnothing$	$[a_2/b_1, +\infty)$	$(-\infty, a_2/b_2] \cup [a_2/b_1, +\infty)$	$(-\infty, a_2/b_2]$	$[0, +\infty)$	$(-\infty, a : 2/b_2] \cup [0, +\infty)$	$(-\infty,0] \cup [a_2/b_1, +\infty)$	$(-\infty,0]$	$(-\infty, +\infty)$
$(-\infty, a_2], a_2 > 0$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, +\infty), a_1 < 0$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$
$[a_1, +\infty), a_1 > 0$	$\varnothing$	$(-\infty, a_1/b_1]$	$(-\infty, a_1/b_1] \cup [a_1/b_2, +\infty)$	$[a_1/b_2, +\infty)$	$(-\infty,0]$	$(-\infty,0] \cup [a_1/b_2, +\infty)$	$(-\infty, a_1/b_1] \cup [0, +\infty)$	$[0, +\infty)$	$(-\infty, +\infty)$
$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$

Table 2.4: Definition for standard interval division with the divisor containing zero and consideration of infinity[13].

Division $0 \notin B$	$[b_1,b_2]$ $b_2 < 0$	$[b_1,b_2]$ $b_1 > 0$	$(-\infty, b_2]$ $b_2 < 0$	$[b_2, +\infty)$ $b_1 > 0$
$[a_1,a_2], a_2 \leq 0$	$[a_2/b_1, a_1/b_2]$	$[a_1/b_1,a_2/b_2]$	$[0, a_1/b_2]$	$[a_1/b_1,0]$
$[a_1,a_2], a_1 \leq 0 \leq a_2$	$[a_2/b_2,a_1/b_2]$	$[a_1/b_1, a_2/b_1]$	$[a_2/b_2,a_1/b_2]$	$[a_1/b_1, a_2/b_1]$
$[a_1,a_2], a_1 \geq 0$	$[a_2/b_2, a_1/b_1]$	$[a_1/b_2,a_2/b_1]$	$[a_2/b_2,0]$	$[0, a_2/b_1]$
[0,0]	[0,0]	[0,0]	[0,0]	[0,0]
$(-\infty, a_2], a_2 \leq 0$	$[a_2/b_1, +\infty)$	$(-\infty, a_2/b_2]$	$[0, +\infty)$	$(-\infty,0]$
$(-\infty, a_2], a_2 \geq 0$	$[a_2/b_2, +\infty)$	$(-\infty, a_2/b_1]$	$[a_2/b_2, +\infty)$	$(-\infty, a_2/b_1]$
$[a_1, +\infty), a_1 \leq 0$	$(-\infty, a_1/b_2]$	$[a_1/b_1, +\infty)$	$(-\infty, a_1/b_2]$	$[a_1/b_1, +\infty)$
$[a_1, +\infty), a_1 \geq 0$	$(-\infty, a_1/b_1]$	$[a_1/b_2, +\infty)$	$(-\infty,0]$	$[0, +\infty)$
$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$	$(-\infty, +\infty)$

Table 2.5: Definition for standard interval division with a divisor without zero and consideration of infinity [13].

$A = [a_1,a_2]$	$B = [b_1,b_2]$	A/B
$0 \in A$	$0 \in B$	$(-\infty, +\infty)$
$0 \notin A$	$B = [0,0]$	$\varnothing$
$a_2 < 0$	$b_1 < b_2 = 0$	$[a_2/b_1, +\infty]$
$a_2 < 0$	$b_1 < 0 < b_2$	$(-\infty, a_2/b_2] \cup [a_2/b_1, +\infty)$
$a_2 < 0$	$0 = b_1 < b_2$	$(-\infty, a_2/b_2]$
$a_1 > 0$	$b_1 < b_2 = 0$	$(-\infty, a_1/b_1]$
$a_1 > 0$	$b_1 < 0 < b_2$	$(-\infty, a_1/b_1] \cup [a_1/b_2, +\infty$
$a_1 > 0$	$0 = b_1 < b_2$	$[a_1/b_2, +\infty)$

Table 2.6: Distinction of cases for interval division A/B where $A,B \in \mathbb{IR}$ [13].

2.1.2 Midpoint-Radius Interval Arithmetic

In our implementation we use standard interval arithmetic [16]. A different approach is presented in [7], which is based on S.M. Rump's thesis [15]. He defines the intervals not by a lower and an upper bound, but by defining an interval as a tuple center point and a radius. Thus in the *midpoint-radius interval arithmetic* an interval is defined as follows:

Definition 2.1.9 (Midpoint-radius intervals)**.** *Every interval $A = [a_1; a_2]$ can be represented by a midpoint $\tilde{a} = (a_1 + a_2)/2$ and its radius $\Delta_a = (a_2 - a_1)/2$,such that $A = [\tilde{a} - \Delta_a; \tilde{a} + \Delta_a]$. We use the following notation to represent the tuple: $A = \langle \tilde{a}; \Delta_a \rangle$.*

Midpoint-radius interval arithmetic is not defined for unbounded intervals. It is impossible to transform an unbounded standard interval to in interval in midpoint-radius interval arithmetic:

$$A = (-\infty; 0] \ , \ \tilde{a} \overset{\text{Def.2.1.9}}{=} \frac{-\infty + 0}{2} = -\infty \ , \ \Delta_a \overset{\text{Def.2.1.9}}{=} \frac{0 + \infty}{2} = \infty \ \text{ but, } A \neq \langle -\infty; \infty \rangle$$

Midpoint-radius interval arithmetic is also not defined to represent strict bounded intervals as they are defined in standard interval arithmetic. In the following we define the addition, subtraction and multiplication for intervals in midpoint-radius arithmetic. There is no need to present the definition of division in midpoint-radius interval arithmetic, as we do not use it in this thesis and the midpoint-radius arithmetic is not used in our implementation. Though the complete definition of the arithmetic can be found in Rumps thesis [15].

We define the addition of two intervals in midpoint-radius interval arithmetic as follows:

Definition 2.1.10 (Midpoint-radius interval addition)**.** *The addition of two intervals $A = \langle \tilde{a}; \Delta_a \rangle$ and $B = \langle \tilde{b}; \Delta_b \rangle$ is defined as $A + B = \langle \tilde{c}; \Delta_c \rangle$, where $\tilde{c} = \tilde{a} + \tilde{b}$ and $\Delta_c = \Delta_a + \Delta_b$.*

Example 2.1.5 (Midpoint-radius interval addition)**.** *Adding the intervals $\langle 4; 3 \rangle$ and $\langle 2; 1 \rangle$ yields to:*

$$\langle 4; 3 \rangle + \langle 2; 1 \rangle = \langle 4 + 2; 3 + 1 \rangle = \langle 6; 4 \rangle$$

Next we present the definition of interval subtraction in midpoint-radius interval arithmetic:

Definition 2.1.11 (Midpoint-radius interval subtraction). *The subtraction of two intervals* $A = \langle \tilde{a}; \Delta_a \rangle$ *and* $B = \langle \tilde{b}; \Delta_b \rangle$ *is defined as* $A - B = \langle \tilde{c}; \Delta_c \rangle$*, where* $\tilde{c} = \tilde{a} - \tilde{b}$ *and* $\Delta_c = \Delta_a + \Delta_b$*.*

Example 2.1.6 (Midpoint-radius interval subtraction). *Subtracting two intervals* $\langle 2; 6 \rangle$ *and* $\langle 3; 2 \rangle$ *yields to:*

$$\langle 2; 6 \rangle - \langle 3; 2 \rangle = \langle 2 - 3; 6 + 2 \rangle = \langle -1; 8 \rangle$$

Finally a definition of the multiplication of two midpoint-radius intervals is provided:

Definition 2.1.12 (Midpoint-radius interval multiplication). *The multiplication of two intervals* $A = \langle \tilde{a}; \Delta_a \rangle$ *and* $B = \langle \tilde{b}; \Delta_b \rangle$ *is defined as* $A \cdot B = \langle \tilde{c}; \Delta_c \rangle$*, where* $\tilde{c} = \tilde{a} \cdot \tilde{b}$ *and* $\Delta_c = |\tilde{a}|\Delta_b + |\tilde{b}|\Delta_a + \Delta_a\Delta_b$*.*

Example 2.1.7 (Midpoint-radius interval multiplication). *Multiplying the intervals* $\langle 4; 2 \rangle$ *and* $\langle 2; 3 \rangle$ *yields to:*

$$\langle 4; 2 \rangle \cdot \langle 2; 3 \rangle = \langle 4 \cdot 2; |4| \cdot 3 + |2| \cdot 2 + 2 \cdot 3 \rangle = \langle 8; 22 \rangle$$

2.1.3 Wrapping Effect

After defining the basic arithmetic operations of both interval arithmetics, we present an effect both arithmetics share. The wrapping effect, also known as the dependency problem, is one of the greatest causes for over-approximation within interval arithmetics. It can occur, in case a reflexive mathematical operation $(x \cdot x)$ or $(x - x)$ is treated like a non-reflexive operation, which can cause a wider solution space. Let us consider the following equations:

$$x = [-1; 1], \quad x^2 = x \cdot x = [-1; 1] \cdot [-1; 1] \overset{\text{Def.2.1.7}}{=} [-1; 1] \tag{2.2}$$

$$x - x = [-1; 1] - [-1; 1] \overset{\text{Def.2.1.6}}{=} [-2; 2] \tag{2.3}$$

In both cases the enclosure is wider than expected, knowing that x^2 is nonnegative and $x - x$ is always zero ([0,0]). As a consequence of the interval arithmetic we use, negative lower bounds can remain negative after squaring and a variable cannot be canceled out by subtracting it with itself.

When squaring an interval we square each element within the set, that is represented by the interval. When multiplying two equal intervals, the resulting set contains all possible multiplications between the elements of both intervals.

Example 2.1.8 (Squaring and multiplication of intervals). *Consider the two intervals* I_A *and* I_B *representing two sets* A,$B \subset \mathbb{N}$*, with* $I_A = I_B = [-1; 1]$

$$I_A^2 = \{-1 \cdot -1, 0 \cdot 0, 1 \cdot 1\} = [0; 1]$$

whereas,

$$I_A \cdot I_B = \{-1 \cdot -1, -1 \cdot 0, -1 \cdot 1, 0 \cdot 0, 0 \cdot 1, 1 \cdot 1\} = [-1,1]$$

Example 2.1.9 (Extinction and subtraction of intervals). *Consider the two intervals* I_A *and* I_B *representing two sets* A,$B \subset \mathbb{N}$*, with* $I_A = I_B = [-1; 1]$

$$I_A - I_A = \{-1 - (-1), 0 - 0, 1 - 1\} = [0; 0]$$

whereas,

$$I_A - I_B = \{-1 - (-1), -1 - 0, -1 - 1, 0 - (-1), 0 - 0, 0 - 1, 1 - (-1), 1 - 0, 1 - 1\} = [-2; 2]$$

These over-approximations occur when calculating with variables within interval arithmetic. Note that the following equations are not affected by the wrapping affect, because they involve two variables.

$$x = y = [-1; 1], \quad x \cdot y = [-1; 1] \cdot [-1; 1] \overset{\text{Def.2.1.7}}{=} [-1; 1]$$

$$x - y = [-1; 1] - [-1; 1] \overset{\text{Def.2.1.6}}{=} [-2; 2]$$

Wrapping within multiplication

In order to minimize the wrapping effect while multiplying intervals, a method to exponentiate intervals is provided by [14].

$$x = [-1; 1], \quad x^2 = (x \cdot x) \cap [0; +\infty] = [0; 1] \tag{2.4}$$

As this arithmetic operation has the potential to reduce the solution space tremendously, we aim to use it as often as possible. If possible we will try to rearrange the Horner scheme in order to use the exponentiation more often.

Example 2.1.10 (Wrapping within multiplication)**.** *Let us consider the following example:*

$$f(x) = 5x^6 + 7x^4 + 15x^2$$

We transform $f(x)$:

$$f(x) = \underbrace{x\Big(x}_{(2.2)}\big(15 + \underbrace{x\big(x}_{(2.2)}\ (7 + \underbrace{x(x}_{(2.2)}(5))\ )\big)\Big) \tag{2.5}$$

With $x = [-2; 5]$ the transformed Term 2.5 will result in $f(x) = [-33150; 82875]$.

We reduce the problem of over-approximation caused by wrapping, by combining as many variables as possible in order to use the exponentiation instead of multiplication.

$$f'(x) = x^2\big(15 + x^2(7 + x^2(5))\big) \tag{2.6}$$

With $x = [-2; 5]$ the transformed Term 2.6 will result in $f(x) = [0; 82875]$.

Wrapping within subtraction

We do not need to consider subtraction related wrapping effects similar to Equation 2.3, as the ICP module uses normalized constraints as input [9]. Normalized constraints have the following form:

$$a_1 \underbrace{x_{1,1}^{e_{1,1}} \cdot \ldots \cdot x_{1,k_1}^{e_{1,k_1}}}_{m_1} + \ldots + a_n \underbrace{x_{n,1}^{e_{n,1}} \cdot \ldots \cdot x_{n,k_n}^{e_{n,k_n}}}_{m_n} + d \sim 0$$

Where $n \geq 0$, a_i and d is an integral number ($\neq 0$), x_{i,j_i} is a real or integer valued variable and e_{i,j_i} is a natural number greater zero (for all $1 \leq i \leq n$ and $1 \leq j_i \leq k_i$). In addition $x_{i,j_i} \neq x_{i,l_i}$ if $j_i \neq l_i$ (for all $1 \leq i \leq n$ and $1 \leq j_i \leq k_i$) and $m_{i_1} \neq m_{i_2}$ if $i_1 \neq i_2$ (for all $1 \leq i_1, i_2 \leq n$). $\sim$ is either $=$ or $<$ and in case n is zero, d is also zero.

Example 2.1.11 (Normalized constraints)**.** *The following constraints are given:*

$$c_1 : x + 3x^2 - x - 2x^2 + 2 = 0$$
$$c_2 : xy^2 - 3x^2 - y^2 + 2x^2 + 2y^2 + x^2 = 0$$

Transforming c_1 and c_2 to normalized constraints leads to:

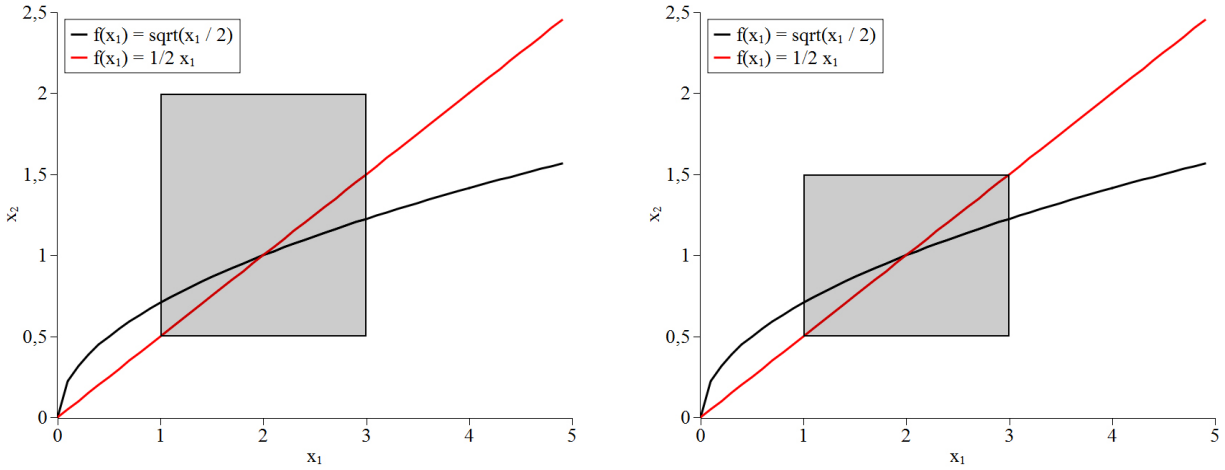
$$c_1 : x^2 + 2 = 0$$
$$c_2 : xy^2 + y^2 = 0$$

A normalized constraint cannot contain a term of the form $ax_i^e - bx_i^e$, with a and b being integral numbers and e being a natural number greater zero. Therefore we do not have to consider the subtraction related wrapping effect .

2.2 Propagation

After introducing the basic characteristics of the arithmetic, we present the mathematical processes of the propagation method, that we use to improve the contraction procedure within the ICP module.

Interval propagation is a method for contracting the solution domains of variables regarding QFNRA equations. As input we take a set of constraints and a search space, represented by an interval box (See Definition



(a) Initial search space

(b) Contracted search space

Figure 2.1: Two graphs displaying the contraction of the search space, before and after applying interval contraction.

2.1.4). This method is only applicable if the input equations (constraints) are at least solvable for one variable. Due to preprocessing the equations either have the form $a \cdot x + h = 0$, with $a \in \mathbb{Q}$ and h being a polynomial not containing x or the from $x^i \cdot m - y = 0$, with m being a monomial not containing x and y being a variable different from x.

The elements within the interval box $B = ([x]_{x_1}, ..., [x]_{x_n})$ are assigned to the variables $x_i, ..., x_n$ of the input equation. In order to conduct interval propagation we need to choose a tuple of an equation and the variable which we want to contract. We call this tuple *contraction candidate*.

Definition 2.2.1 (Contraction Candidate)**.** *A contration candidate c is a tuple*

$$c = \langle e, x_i \rangle$$

where e is an equation and x_i is an variable occurring in e.

After choosing a contraction candidate, we solve the input equation e for x_i. All variables except x_i are replaced by their respective interval domains from B. Next we calculate the solution of the equation and get a new interval $[y]$ as a result. Finally we intersect $[y]$ with $[x]_{x_i}$ and get $[z]$. Then we replace $[x]_{x_i}$ with $[z]$. Because interval arithmetic is inclusion isotonic, we can guarantee, that $[z]$ is containing the solution, if the solution is also contained in $[x]_{x_i}$. Consequently if the intersection returns an empty set, the solution is not contained in $[x]_{x_i}$.

Example 2.2.1. *Let x_1 and x_2 be variables and $c_1 : 2x_2 - x_1 = 0$ and $c_2 : 2x_2^2 - x_1 = 0$ be a set of constraints.*
$[x] = ([x]_{x_1}, [x]_{x_2}) \in \mathbb{IR}^2$,
where $[x]_{x_1} = [1; 3]$, $[x]_{x_2} = [0.5; 2]$, $c_1 : 2x_2 - x_1 = 0$, $c_2 : 2x_2^2 - x_1 = 0$ The initial search space is displayed in Figure 2.1a. We choose the contraction candidate: $\langle c_1, x_1 \rangle$ Solving c_1 for x_2 yields:

$$[y] = \frac{1}{2}[1,3] = [0.5,\ 1.5]$$

Intersect the result with $[x]_{x_1}$:

$$[z] = [0.5; 2] \cap [0.5;\ 1.5] = [0.5;\ 1.5]$$

And replace $[x]_{x_1}$ by $[z]$. The contracted search space is shown in Figure 2.1b.

2.3 Newton's Method

Next we present the second contraction method implemented within the ICP module. Newton's method, also known as Newton-Raphson method provides a procedure to iteratively approximate the root of a univatiate function. We will use this procedure in order to approximate solutions for the constraints. Let f be a real-valued

function of a real variable x, and f be a continuously differentiable. Recall the univariate Newton-Raphson method: We repeat the following step,

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

until the desired accuracy is reached. (See Figure 2.2)

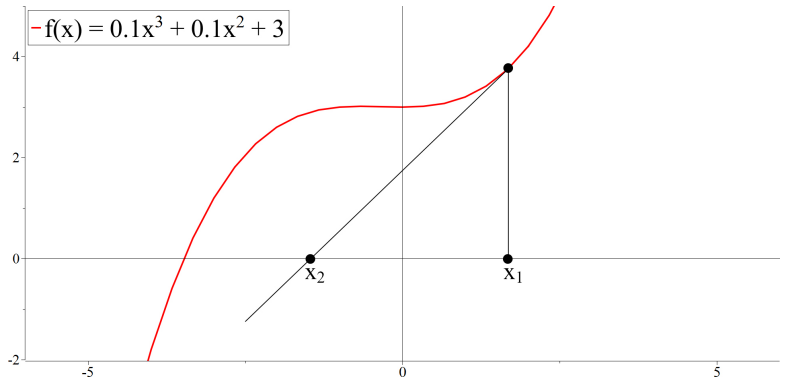


Figure 2.2: Univariate Newton iteration with $f(x) = \frac{1}{10}x^3 + \frac{1}{10}x^2 + 3$ at x_1

We can apply a similar method for interval-valued multivariate polynomials. The following method is based on a approach initially presented by Ramon E. Moore [14]. As input we get an n-dimensional function $f(x_1, ..., x_n)$. When assigning a value to each variable except for x_j we get the one-dimensional function f_i. The newton operation for intervals is defined by [13] as follows:

Definition 2.3.1 (Newton operator for intervals)**.** *Let $D \subseteq \mathbb{R}^n$, $f : D \to \mathbb{R}^n$, $f = (f_1, ..., f_n)^T$ be a continuously differentiable function, and let $[x] = ([x]_{x_1},...,[x]_{x_n})^T \in \mathbb{IR}^n$ be an interval vector with $[x] \subseteq D$ and $i,j \in \{1, ..., n\}$. Then component-wise interval Newton operator N_{cmp} is defined by:*

$$N_{cmp}([x],i,j) = N(\underbrace{[x]}_{box}, \underbrace{f_i(x_1, ..., x_n), j}_{ContractionCandidate}) := c_j - \frac{f_i([x]_{x_1},...,[x]_{x_{j-1}}, c_j, [x]_{x_{j+1}},...,[x]_{x_n})}{\frac{\partial f_i}{\partial x_j}([x]_{x_1},...,[x]_{x_n})}$$

By choosing a point c_j within the interval we want to contract, usually the center point, we can apply the Newton operator on our contraction candidate and intersect its result with the original interval. As Kulisch proves in his thesis ([13]) in case the solution x^* is within the box $[x]$ it has to be in $N_{cmp}([x],i,j)$ as well. Therefore in case $[x] \cap N_{cmp}([x],i,j) = \emptyset$ there is no solution in $[x]$. The inclusion isotonicity of interval arithmetic and the N_{cmp} operator guarantee, that we do not cut off a solution, while contracting.

Example 2.3.1 (Newton method)**.** *Let x_i be a variable and c_i a constraint. $[x] = ([x]_{x_1}, [x]_{x_2}) \in \mathbb{IR}^2$) where $[x]_{x_1} = [-2,0]$, $[x]_{x_2} = [-1;1]$, $c_1 : 1/2x^3 - x + 1 = 0$. We chose c_1 and variable x_1 as the first contraction candidate.*

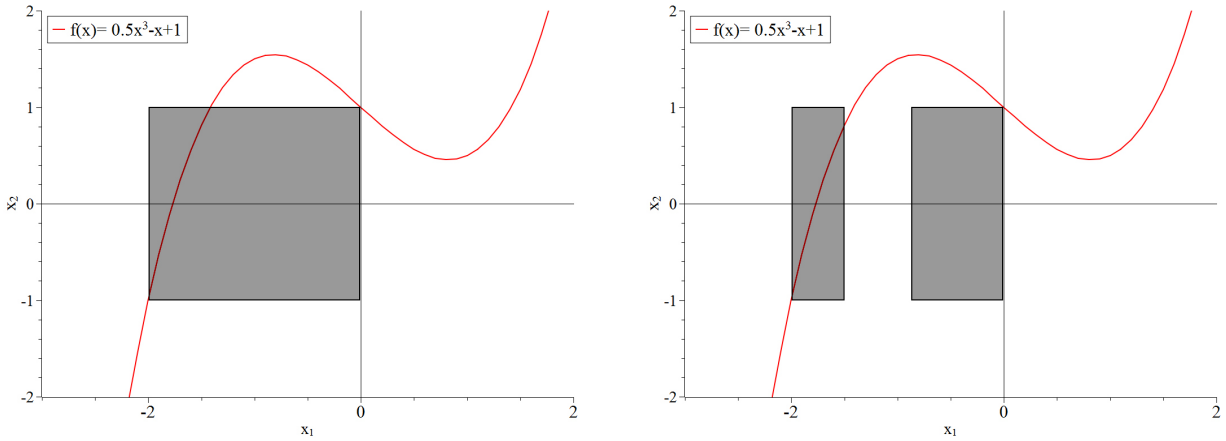
$$f(x) = \frac{1}{2}x^3 - x + 1$$
$$f'(x) = \frac{3}{2}x^2 - 1$$

With center point $c_1 = -1$

$$N_{cmp}([x],f(x_1),x_1) = -1 - \frac{\frac{1}{2}(-1)^3 - 1 + 1}{\frac{3}{2}[-2;0]^2 - 1} = -1 + \frac{1/2}{[-1;8]}$$

With Definition 2.1.8 follows:

$$N_{cmp}([x],f(x_1),x_1) = -1 + \left((-\infty; -\frac{1}{2}] \cup [\frac{1}{16}; +\infty)\right) = (-\infty; -\frac{3}{2}] \cup [-\frac{15}{16}; +\infty)$$



(a) Initial search space (b) Contracted search space after split occurred

Figure 2.3: Two graphs displaying the contraction of the search space, before and after applying the Newton method for contraction.

Intersect with original search space for x_1, to get the contracted search space:

$$[-2;0] \cap \left((-\infty;-\frac{3}{2}] \cup [-\frac{15}{16};+\infty)\right) = [-2;-\frac{3}{2}] \cup [-\frac{15}{16};0]$$

The resulting search space is significantly smaller and split in two parts, due to interval division (see Figure 2.3).

2.4 QFNRA formulas

The ICP method we aim to improve is an incomplete method, that proves to be a powerful tool to aid solving *quantifier-free non-linear real arithmetic* (QFNRA) problems.

Definition 2.4.1 (QFNRA formulas)**.** *Quantifier-free nonlinear real formula are formed by the following grammar:*

$$\begin{aligned} p &:= r \mid x \mid (p+p) \mid (p-p) \mid (p \cdot p) \\ c &:= p < 0 \mid p = 0 \\ \varphi &:= c \mid (\varphi \wedge \varphi) \mid \neg\varphi \end{aligned}$$

where $x \in Var(\varphi)$ represents a variable, $Var(\varphi) = \{x_1, \cdots, x_n\}$ is a set of all variables occurring in φ and $r \in \mathbb{Q}$ is a rational constant. p is a polynomial, c is a constant and φ is a formula of QFNRA.

Syntactic sugar like $(p \leq 0)$, $(p > 0)$, $(p \geq 0)$ and $(\varphi \vee \varphi)$ can be derived from this grammar. A polynomial p is defined as a sum of terms, which is a coefficient r ($r \in \mathbb{Q}$) times a monomial. A monomial is a product of powers of variables x_j with nonnegative exponents e. Thus we can rewrite a polynomial p in QFNRA as follows:

$$p := \sum_{i=1}^{n} r_i \prod_{j=0}^{n_i} x_j^{e_{ij}}$$

There are currently only a few approaches for satisfiability-checking for QFNRA formulas. Solvers for QFNRA have an doubly exponential worst case complexity [6]. Known approaches are the cylindrical algebraic decomposition (CAD) [8] which is a complete method, virtual substitution (VS) [1] and Gröbner bases [2], which are both incomplete, meaning these methods are not able to determine the satisfiability for every input formula.

2.5 Satisfiability Modulo Theories Solving

Satisfiability modulo theories (SMT) solving provides an algorithmic framework that allows checking the satisfiability of Boolean combinations of theory constraints of the theory of QFNRA. SMT solvers consist of a combination of a SAT solver and one or more theory solvers.

The SMT solver (Figure 2.4) starts by creating a Boolean abstraction of the formula φ, which is brought to a conjunctive normal form (CNF). For every constraint in the original formula, a fresh Boolean variable is introduced by the Boolean abstraction, consequently the Boolean skeleton of the original formula is kept intact. Next, the SAT solver tries to assign the variables of the Boolean abstraction. Each new introduced Boolean variable represents a QFNRA subformula from the original input formula. These subformulas are constraints. After each partial assignment, a set of asserted subformulas whose respective Boolean variables have been assigned, is checked for consistency with the Boolean assignment, by the theory solver.

Due partial assignment, the SMT solver is called less-lazy SMT solver. An alternative to less-lazy SMT is the full-lazy SMT solver, which assigns all variables at once and then passes the corresponding constraints to the theory solver. Next the theory solver attempts to find a solution for the subset of given constraints. In case the theory solver finds a solution for the subset, it hands back the information to the SAT solver, such that the SAT solver can assign further variables. If the theory solver can find a solution for the assigned variables, it will declare the subset as satisfiable. If the theory solver can not find a solution it will return an explanation for the SAT solver in the form of an infeasible subset of the given constraints. The information of the infeasible subset is used to resolve the conflict and assign the variables differently. In case the SAT solver cannot find an assignment for the Boolean structure, it declares the input formula as unsatisfiable. The SMT-solver repeats this process until all variables are successfully assigned, or a conflict occurs, that can not be resolved.

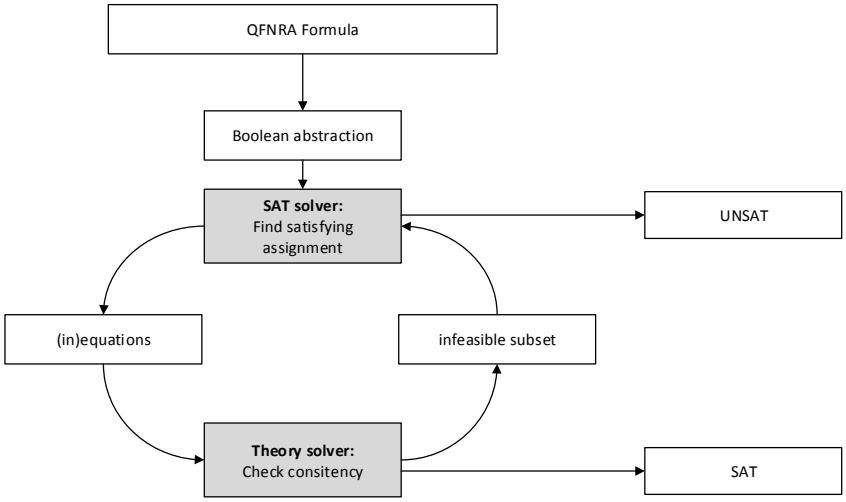


Figure 2.4: Diagram displaying the basic architecture of SMT solving

2.5.1 SMT-RAT

The implementations we realize in this thesis are presented within the context of the preexisting toolbox SMT-RAT [3]. SMT-RAT provides a framework to exchange, combine and rearrange different modules to form an SMT solver according to a user-defined strategy. Among others some modules implement Gröbner bases, virtual substitution, cylindrical algebraic decomposition and interval constraint propagation [9],[3]. We can summarize the architecture of this toolbox into three parts: *modules*, a *manager* and a *strategy*.

Each *module* encapsulates an approach to process a set of constraints. The modules share a common interface. Each module m initially starts with an empty set of received formulas $C_{rec}(m)$. We are able to add and remove formulas to $C_{rec}(m)$ with dedicated functions, that are implemented in each module. The essential function of a module is the consistency check. It decides whether the conjunction of received formulas in $C_{rec}(m)$ is satisfiable or not. In case the module declares the $C_{rec}(m)$ as unsatisfiabe, it is able to return UNSAT and an infeasible subset $C_{inf}(m) \subseteq C_{rec}(m)$ otherwise it can return SAT or UNKNOWN. Besides modules can specify lemmas, that encapsulate information from the internal state of a module that can be propagated among other modules. Modules themselves can ask other modules to determine the satisfiability of a set of formulas $C_{pas}(m)$. In case a module passes a set of formulas, the manager will divert the (sub-)set to a module that might be more suited to determine the satisfiability. Note that in SMT-RAT the SAT solver is also implemented as a module.

The *strategy* determines the order in which each module is used. The strategy is defined by the user and enables him to create powerful and efficient *composed theory solvers*. A strategy consists of condition-module pairs, which allows to define a sequence of modules based on properties of the input formulas.

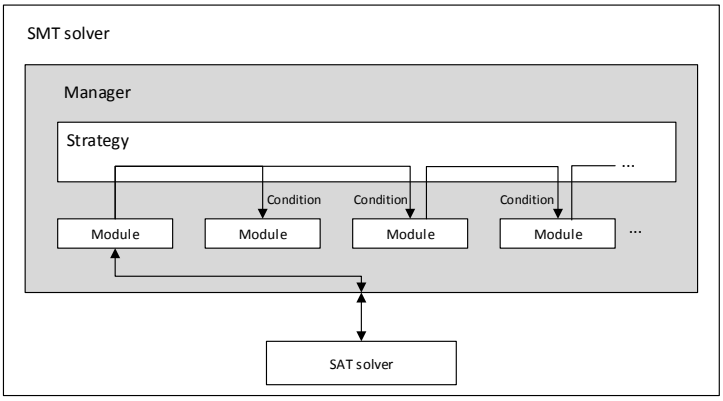
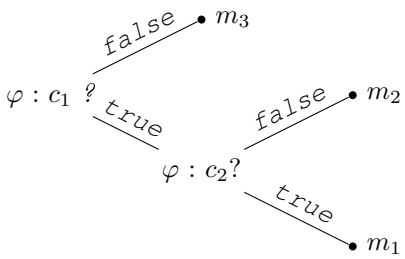


Figure 2.5: Diagram displaying the relation between *modules*, *strategy* and *manager*, within the SMT-RAT framework, based on [3], [9].

Example 2.5.1 (SMT-RAT strategy with conditions)**.** *Let φ be input of the SMT-Solver, and $c_1?(c_2?m_1 : m_2) : m_3$ the Strategy, where c_1 and c_2 are conditions of φ and m_1, m_2 and m_3 are modules theory solver modules of SMT-Rat*



If φ fulfills c_1 there is a check whether φ fulfills c_2. If it fulfills c_2 as well m_1 will be used as a theory solver, otherwise m_2 will try to check φ. If φ does not fulfill c_1, m_3 will be used as a theory solver.

It is also possible to define a strategy without any conditions:

Example 2.5.2 (SMT-RAT strategy)**.** *One of the strategies available for SMT-RAT (`RatTwo`): This strategy implements no conditions, as it only has a serial sequence of modules.*

$$\rightarrow CNF_M \rightarrow PP_M \rightarrow SAT_M \rightarrow ICP_M \rightarrow VS_M \rightarrow CAD_M$$

CNF_M transforms the input formula C_{input} to a conjunctive normal form. PP_M performs preprocessing. SAT_M is a SAT solver that abstracts the received formula $C_{rec}(SAT_M)$ to propositional logic and checks for satisfiability. It passes on the constraints which are abstracted by Boolean variables. ICP_M is a ICP procedure, that can lift splitting decisions and contraction lemmas to SAT_M. VS_M is a module implementing virtual substitution, and CAD_M is a module implementing cylindrical algebraic decomposition.

The third part is the *manager*, it keeps track of all the modules and manages the communication between each module. It governs the strategy and it also holds the SMT solver's inital input C_{input}.

2.6 Interval Constraint Propagation

In this section we present the ICP module as implemented by S. Schupp [16]. In the context of the SMT solver the main task of the ICP module is to contract the solution space up to a dedicated threshold. If possible the module will return *UNSAT*. Note that the ICP module will not return *SAT*, as ICP can only exclude solutions.

After contracting, the module passes its results to the next solver and serves this way as a preprocessor.

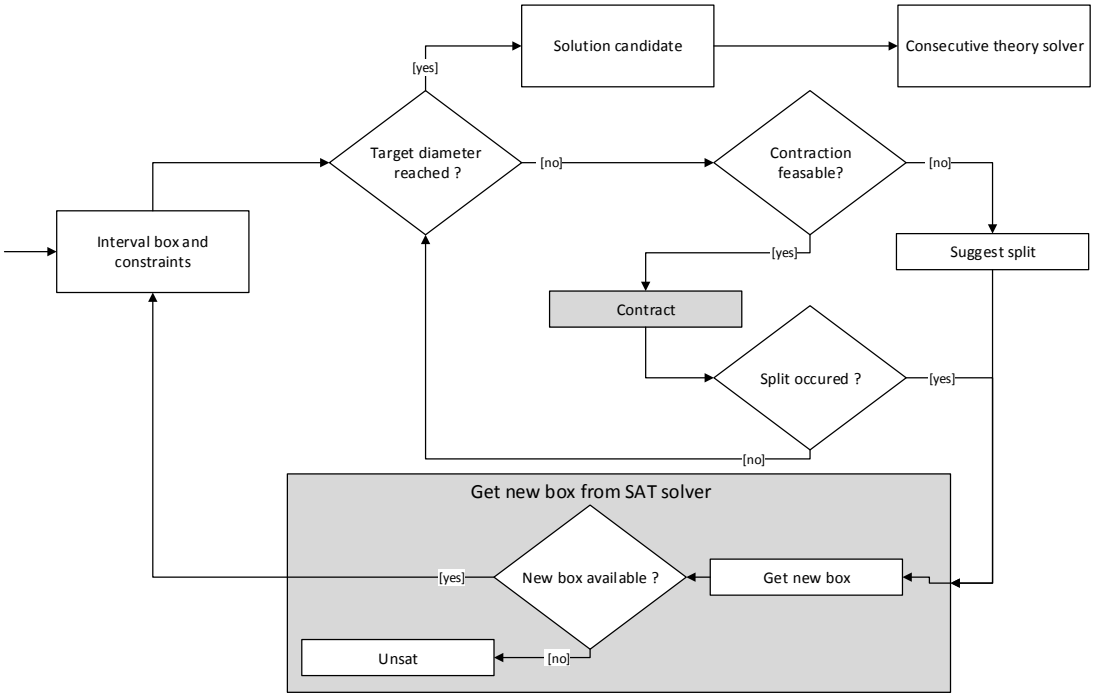


Figure 2.6: Diagram displaying the contraction procedure of the ICP module.

2.6.1 Algorithm

The idea behind ICP is the reduction of the given search space for a set of constraints until a specified precision is reached. ICP achieves this by repeatedly choosing contraction candidates (see Definition 2.2.1) and applying contraction methods (see Section 2.7) at the search space.

The detailed process of the algorithm is as follows (see Figure 2.6): We take an interval box and a set of constraints as input. In case an interval box has reached the target diameter, or is even smaller, we will pass it as a solution candidate to the next solver, as a solution candidate. In case the target diameter is not reached yet, we choose a new contraction candidate. In case there is no viable contraction candidate, we split the search space and give that information to the SAT solver, which tries to provide a new box. We call this splitting an *autonomous split*. In case there is a viable contraction candidate, we perform a contraction. During that contraction a heteronomous split might occur (see Definition 2.1.8). In case a split occurred we inform the SAT solver which chooses a new box. After contracting we check whether the target diameter is reached and continue contracting until this is the case. During the ICP process the SAT solver might be asked to conduct a (heteronomous or autonomous) split and provide a new Box. If it cannot provide a new box it declares the input set as unsatisfiable.

2.6.2 Preprocessing

In order to be able to contract, the module requires constraints that are solvable for a single variable. Therefore it first separates linear and nonlinear constraints, and then linearizes all nonlinear constraints by introducing new variables and constraints for nonlinear monomials [16].

Example 2.6.1 (Linearization)**.** *Non-linearized Input:*

$$x_1^3 \cdot x_2 + x_3 = 0$$

Linearized output with new variable for nonlinear monomial:

$$v_1 + x_3 = 0 \wedge v_1 = x_1^3 \cdot x_2$$

In order to enable the effective transformation into Horner Schemes, we add the original subformula to every new created constraint. Because ICP is known to suffer from the slow convergence problem [16], [17] when used to solve a set of linear constraints and because there are more efficient solvers to handle linear constraints, we

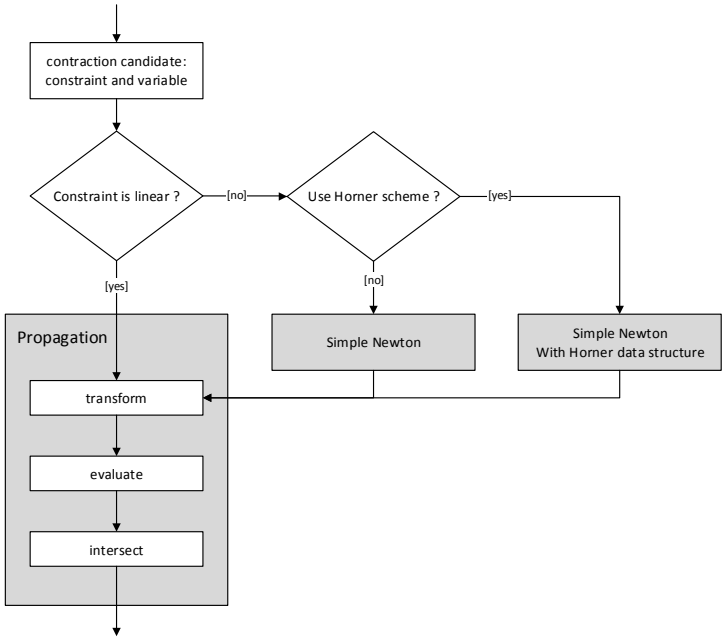


Figure 2.7: Diagram displaying the heuristic within the ICP module that determines the contraction method.

will pass the set of linear constraints to a *linear real arithmetic* (LRA) solver. The set of nonlinear constraints is passed on to be contracted by the ICP module. Note that all variables used in the ICP module are required to be bounded by intervals, which if needed are set up by an initial box. In case the ICP module returns UNSAT for the initial box, another solver is used to check the satisfiability for the set of constraints, with the remaining search space outside the initial box.

2.7 Contraction

One of the key functions of the ICP solver is contraction. Additionally to the preexisting interval based Newton method a propagation method was added along with a new data structure for the Newton method (see Figure 2.7). Basic interval propagation provides a cheap method to contract the search space of constraints. Its contractions cost in average less computation time, than the contractions of the Newton method, but the effect of the contractions is greater as well. In the following sections we present the central ideas and processes for each contraction method.

Before contracting a contraction candidate is chosen, thus each contraction method knows the current interval box and the set of constraints.

2.7.1 Propagation

Interval propagation is a cheap and powerful method to contract the search space (see Section 2.2), within ICP this procedure is used as follows: Propagation can be divided into three parts (see Figure 2.7). In the first part we solve (*transform*) the input constraint for one variable. Thanks to preprocessing we only have to deal with two types of constraints: Case 1:

$$a \cdot x + h = 0 \Leftrightarrow x = -\frac{h}{a}$$

where $a \in \mathbb{Q}$ and h is a polynomial not containing x and not having a constant part. The variable x is determined by the contraction candidate. Case 2:

$$x^i \cdot m - y = 0 \Leftrightarrow x = \sqrt[i]{\frac{y}{m}}$$

where m is a monomial not containing x and y is a variable different from x. The variable x is determined by the contraction candidate.

In the second part we *evaluate* the term created in the first part. We solve it, by using interval arithmetic and inserting the bounds of all variables except x. An heteronomous split might occur, in case we have to calculate a division (see Definition 2.1.8).

In the third part we *intersect* the result(s) from the evaluation with the original input. As a result there might also be two intervals. Based on the amount of intervals we have at the end, either one or two, we have to get a new box from the SAT solver.

The newly introduced propagation method proves to accelerate the contraction process and thereby leads to a higher efficiency and a greater success rate of the SMT-Solver. The results will be presented in Section 5.2.

2.7.2 Newton's Method

The second contraction method implemented in the ICP module, is the contraction based on Newton's method. The arithmetic properties of the procedure are presented in Section 2.3.

Within the ICP module, this procedure provides the polynomial and its derivative. Depending on the setting of the ICP module, the procedure transforms both the derivative and the polynomial into a multivariate Horner scheme, or continues calculating with a multivariate polynomial (See Figure 2.7).

2.7.3 Precision

Within the ICP module, the precision of the result is defined by the user. The module contracts as long as the target diameter is not reached. Choosing a good target diameter is crucial to the performance of the SMT solver. It determines the amount of preprocessing the ICP module is doing for the consecutive solver. Selecting a small target diameter will require more contraction by the ICP while choosing a greater diameter will pass on more work to the next solver. Depending on the following solver, a good diameter that does a good trade-off between both solvers should be chosen.

Chapter 3

Horner Schemes

In this section we present the characteristics and advantages of Horner schemes. First we will define Horner schemes, before taking a closer look at the benefits of evaluating polynomials that were transformed into Horner schemes. We will focus on its effects on efficiency and precision, and finally we will take a look at two heuristics for the construction of a Horner scheme.

One of the most efficient ways to evaluate univariate polynomials, is to use a Horner scheme. Univariate Horner schemes provide a transfomation for a polynomial that contains the minimal amount of mathematical operations possible. There is only one procedure for the creation of a univariate Horner scheme, therefore each univariate polynomial has only one unique Horner scheme. We define univariate Horner schemes based on [18].

Definition 3.0.1 (Univariate Horner scheme)**.** *Given the polynomial*

$$p(x) = a_0 + a_1x + a_2x^2 + ... + a_nx^n$$

With the coefficients $a_0, ..., a_n$ *,we simply can transform* p *to the univariate Horner scheme*

$$p(x) = a_0 + x(a_1 + x(a_2 + ... + x(a_{n-1} + xa_n)...))$$

Our ICP implementation is mostly using *multivariate* polynomials, therefore we need to define *multivariate Horner schemes*. In contrast to univariate Horner schemes, there is not a unique procedure to create a multivariate Horner scheme. It is possible to transform a multivariate polynomial into different multivariate Horner schemes, which will differ in the amount of arithmetic operations. In Section 3.3 we present two strategies for this process. The definition is similar to the one of univariate Horner schemes:

Definition 3.0.2 (Multivariate Horner scheme)**.** *The multivariate Horner scheme* H *is defined as follows:*

$$\begin{aligned} \forall\ a \in \mathbb{Q} &: a \in H \\ \forall\ h_{dep} \in H &: x_i^e \cdot (h_{dep}) \in H \\ \forall\ h_{dep}, h_{ind} \in H &: x_i^e \cdot (h_{dep}) + h_{ind} \in H \end{aligned}$$

where $e \in \mathbb{N} \setminus \{0\}$ *and the variable* x_i *does not occur in* h_{ind}*.*

The structure in which we embed the Horner schemes consists of 6 components. Every Horner scheme instance that is not constant has a *variable* x_i and a associated *exponent* e. We call the part which will be multiplied with that variable *dependent-part* (h_{dep}) and the part which we add up without multiplying it with the variable *independent-part* (h_{ind}). Both dependent and independent part are Horner schemes themselves. Both dependent and independent part have an associated *coefficient* a.

3.1 Effects on Efficiency

Horner schemes are known for their properties of minimizing the arithmetic operations of *univariate* polynomials [7], which reduces the cost to evaluate the transformed polynomial.

Example 3.1.1 (Univariate Horner scheme)**.** *If we want to optimize the number of required calculations to solve this polynomial,*

$$f(x) = \underbrace{5 + \underbrace{2x}_{1} + \underbrace{4x^2}_{2} + \underbrace{3x^3}_{3}}_{9\ operations}$$

we can use the Horner scheme to simplify $f(x)$.

$$f'(x) = 5 + x(2 + x(\underbrace{4 + x(3)}_{2})) \quad \underbrace{\qquad\qquad}_{4} \quad \underbrace{\qquad\qquad\qquad\qquad}_{6\ operations}$$

It is well-established that *univariate* Horner schemes decrease the time needed by a computer to solve a polynomial by minimizing the amount of needed arithmetic operations. However it is unclear whether a *multivariate* Horner scheme also returns the absolute minimum of mathematical operations.
Although it is possible for a *multivariate* Horner scheme to represent a polynomial which is not reducible, it is problematic to find heuristics which lead to that form.
The problem lies within the choice of the extracted variable. The amount of mathematical operations is dependent on the order in which we choose to extract variables.

Example 3.1.2 (Multivariate Horner scheme)**.** *Consider the polynomial, which requires 8 arithmetic operations to solve:*

$$f(x_1, x_2, x_3) = \underbrace{\underbrace{x_1x_2^2}_{2} + \underbrace{x_1x_3}_{1} + \underbrace{x_1x_2^2x_3}_{3}}_{8\ operations}$$

extract x_1:

$$f(x_1, x_2, x_3) = x_1(x_2^2 + x_3 + x_2^2x_3)$$

extract x_2 (twice):

$$f(x_1, x_2, x_3) = x_1(x_2 \cdot x_2(1 + x_3) + x_3) = \underbrace{x_1 \cdot (\underbrace{x_2^2 \cdot (1 + x_3)}_{3} + x_3)}_{5\ operations}$$

To solve the form above we need 5 arithmetic operations. However, if we extract variables in a different order, the amount might differ:
Given polynomial (same as above):

$$f(x_1, x_2, x_3) = \underbrace{\underbrace{x_1x_2^2}_{2} + \underbrace{x_1x_3}_{1} + \underbrace{x_1x_2^2x_3}_{3}}_{8\ operations}$$

extract x_1:

$$f(x_1, x_2, x_3) = x_1(x_2^2 + x_3 + x_2^2x_3)$$

extract x_3:

$$f(x_1, x_2, x_3) = \underbrace{\underbrace{x_1 \cdot (x_3}_{1} \cdot \underbrace{(1 + x_2^2)}_{2} + \underbrace{x_2^2}_{1})}_{6\ operations}$$

The amount of arithmetic operations needed to solve this Horner scheme differ from the amount needed to solve the first Horner scheme.

By reducing the amount of arithmetic operations within the polynomial we reduce the computation time for the evaluation of the polynomial. In order to be able to decrease the number of mathematical operations for a polynomial p, it needs to contain at least two monomials containing the same variable.

Example 3.1.3. *Given are the following polynomials:*
Polynomial only containing pairwise diverse variables:

$$\underbrace{x_1x_2 + x_3x_4 + x_5x_6}_{5\ operations} \Leftrightarrow \underbrace{x_1(x_2) + x_3(x_4) + x_5(x_6)}_{5\ operations} \tag{3.1}$$

We are able to transform this polynomial, but we do not get a smaller number of arithmetic operations.
A polynomial containing only two monomials with the same variable:

$$\underbrace{x_1x_2 + x_1x_3 + x_4x_5}_{5\ operations} \Leftrightarrow \underbrace{x_1(x_2 + x_3) + x_3x_5}_{4\ operations}$$

We can extract one variable from two monomials, and thereby reduce the amount of arithmetic operations by one.
A polynomial containing n monomials, where m monomials share the same variable.

$$\underbrace{x_1x_1 + x_1x_2 + ... + x_1x_m}_{2m-1\ operations} \Leftrightarrow \underbrace{x_1(x_1 + x_2 + ... + x_m)}_{(2m-1)-(n-1)\ operations}$$

The amount of arithmetic operations we are able to save is related to the number of monomials that contain the same variable.

Although Horner schemes can provide a significant reduction of computation time, it requires a certain type of polynomials to be effective. We expect Horner schemes to be more effective on sets of constraints, that contain polynomials with a high amount of terms but a small count of variables. Problems with mainly linear constraints might require the same amount of computation time because they still need to be converted to Horner schemes, but will not benefit from any reduction of arithmetic operations (see Example 3.1.3).

3.2 Effects on Precision

One of the largest over-approximations within interval arithmetic is caused by the dependency problem (Section 2.1.3), but as Paper [7] suggests there is at least one more cause within interval multiplication, that causes a loss of precision. In the following section we demonstrate that we are able to reduce the over-approximation when using Horner schemes to evaluate a polynomial.

3.2.1 Minimizing Over-Approximation:

Based on Martine Ceberio's and Vladik Kreinovich's paper [7] we assume that using Horner schemes will provide a higher precision. In the following we will retrace the proof as it is shown in [7]. The paper demonstrates that transforming a polynomial from an extended form to a Horner-based form will lead to a higher precision when using midpoint-radius interval arithmetic. Therefore the diameter of both resulting intervals $a \cdot (b + c)$ and $ab + bc$ is compared with each other. The following needs to be proven:

$$d(a \cdot (b + c)) \leq d(ab + bc) \tag{3.2}$$

Where the function $d(I)$ provides the diameter $d = \overline{i} - \underline{i}$ of the interval $I = [\underline{i}; \overline{i}]$. First the left side of the inequation is evaluated. Midpoint-radius interval arithmetic is used, therefore the resulting interval is the following (see Definition 2.1.9):

$$a \cdot (b + c) = \langle \tilde{l}\ ;\ \Delta_l \rangle$$

Based on Definition 2.1.12 and Definition 2.1.10 the midpoint of the resulting interval is deducible:

$$\tilde{l} = \tilde{a} \cdot (\tilde{b} + \tilde{c}) \tag{3.3}$$

Based on Definitions 2.1.12 and 2.1.12 the radius of the resulting interval can be calculated:

$$\Delta_l = |\tilde{a}|(\Delta_b + \Delta_c) + |\tilde{b} + \tilde{c}|\Delta_a + \Delta_a(\Delta_b + \Delta_c) \tag{3.4}$$

Next the right side of Inequation 3.2 is evaluated. Both the summands are substituted. The first summand is substituted as follows:

$$a \cdot b = \langle \tilde{e}\ ;\ \Delta_e \rangle$$

Based on Definition 2.1.12 and Definition 2.1.10 the midpoint $\tilde{e}$ and radius Δ_e solve as follows:

$$\tilde{e} = \tilde{a}\tilde{b},\ \Delta_e = |\tilde{a}|\Delta_b + |\tilde{b}|\Delta_a + \Delta_a\Delta_b$$

Analogous, the second summand substitutes to $\tilde{f}$ and Δ_f:

$$a \cdot c = \langle \tilde{f} \ ; \ \Delta_f \rangle$$

where

$$\tilde{f} = \tilde{a}\tilde{c}, \ \ \Delta_e = |\tilde{a}|\Delta_c + |\tilde{c}|\Delta_a + \Delta_a\Delta_c$$

The right side of Inequation 3.2 solves to a new interval with midpoint $\tilde{r}$ and radius Δ_r:

$$a \cdot b + a \cdot c = \langle \tilde{r} \ ; \ \Delta_r \rangle$$

By applying Definition 2.1.10 the result for the right side of the Inequation 3.2 yields:

$$\begin{aligned} \tilde{r} = \tilde{e} + \tilde{f}, \ \ \Delta_r = \Delta_e + \Delta_f \\ \tilde{r} = \tilde{a}\tilde{b} + \tilde{a}\tilde{c}, \ \ \Delta_r = |\tilde{a}|\Delta_b + |\tilde{b}|\Delta_a + |\tilde{a}|\Delta_c + |\tilde{c}|\Delta_a + \Delta_a(\Delta_b + \Delta_c) \end{aligned} \tag{3.5}$$

Comparing the right side (3.4, 3.3) with the left side (3.5), shows that both have the same midpoint, but differ in radius:

$$\Delta_r - \Delta_l = (|\tilde{b}| + |\tilde{c}| - |\tilde{b} + \tilde{c}|)\Delta_a$$

Thus Inequation 3.2 holds. Therefore, when using midpoint-radius interval arithmetic a higher precision is reachable, if we extract a variable from a term. We now try to apply this proof to the standard interval arithmetic, which we use in our implementation:

$$I_1 = A \cdot B + A \cdot C \tag{3.6}$$
$$I_2 = A \cdot (B + C) \tag{3.7}$$

where

$$I_1, I_2, A, B, C \in \mathbb{IR}, A = [a_1; a_2], B = [b_1; b_2], C = [c_1; c_2]$$

We solve (3.6) with Definition 2.1.5 and Definition 2.1.7:

$$\begin{aligned} I_1 = [min\{a_1b_1, a_1b_2, a_2b_1, a_2b_2\}; max\{a_1b_1, a_1b_2, a_2b_1, a_2b_2\}] + \\ [min\{a_1c_1, a_1c_2, a_2c_1, a_2c_2\}; max\{a_1c_1, a_1c_2, a_2c_1, a_2c_2\}] \end{aligned} \tag{3.8}$$

We expand the lower bound $\underline{i_1}$:

$$\begin{aligned} \underline{i_1} = min\{&a_1b_1 + a_1c_1, a_1b_1 + a_1c_2, a_1b_1 + a_2c_1, a_1b_1 + a_2c_2, \\ &a_1b_2 + a_1c_1, a_1b_2 + a_1c_2, a_1b_2 + a_2c_1, a_1b_2 + a_2c_2, \\ &a_2b_1 + a_1c_1, a_2b_1 + a_1c_2, a_2b_1 + a_2c_1, a_2b_1 + a_2c_2, \\ &a_2b_2 + a_1c_1, a_2b_2 + a_1c_2, a_2b_2 + a_2c_1, a_2b_2 + a_2c_2\} \end{aligned}$$

Next we can simplify $\underline{i_1}$:

$$\begin{aligned} \underline{i_1} = min\{&\boldsymbol{a_1(b_1 + c_1)}, a_1(b_1 + c_2), a_1b_1 + a_2c_1, a_1b_1 + a_2c_2, \\ &a_1(b_2 + c_1), \boldsymbol{a_1(b_2 + c_2)}, a_1b_2 + a_2c_1, a_1b_2 + a_2c_2, \\ &a_2b_1 + a_1c_1, a_2b_1 + a_1c_2, \boldsymbol{a_2(b_1 + c_1)}, a_2(b_1 + c_2), \\ &a_2b_2 + a_1c_1, a_2b_2 + a_1c_2, a_2(b_2 + c_1), \boldsymbol{a_2(b_2 + c_2)}\} \end{aligned} \tag{3.9}$$

Analogous to $\underline{i_1}$, we calculate the upper bound $\overline{i_1}$:

$$\begin{aligned} \overline{i_1} = max\{&\boldsymbol{a_1(b_1 + c_1)}, a_1(b_1 + c_2), a_1b_1 + a_2c_1, a_1b_1 + a_2c_2, \\ &a_1(b_2 + c_1), \boldsymbol{a_1(b_2 + c_2)}, a_1b_2 + a_2c_1, a_1b_2 + a_2c_2, \\ &a_2b_1 + a_1c_1, a_2b_1 + a_1c_2, \boldsymbol{a_2(b_1 + c_1)}, a_2(b_1 + c_2), \\ &a_2b_2 + a_1c_1, a_2b_2 + a_1c_2, a_2(b_2 + c_1), \boldsymbol{a_2(b_2 + c_2)}\} \end{aligned} \tag{3.10}$$

We solve (3.7) with Definition 2.1.5 and Definition 2.1.7:

$$\begin{aligned} I_2 = [&min\{a_1(b_1+c_1), a_1(b_2+c_2), a_2(b_1+c_1), a_2(b_2+c_2)\}; \\ &max\{a_1(b_1+c_1), a_1(b_2+c_2), a_2(b_1+c_1), a_2(b_2+c_2)\}] \\ \underline{i_2} = &min\{a_1(b_1+c_1), a_1(b_2+c_2), a_2(b_1+c_1), a_2(b_2+c_2)\} \\ \overline{i_2} = &max\{a_1(b_1+c_1), a_1(b_2+c_2), a_2(b_1+c_1), a_2(b_2+c_2)\} \end{aligned}$$

In order to show, that the inequation $d(A{\cdot}B{+}A{\cdot}C) \geq d(A(B{+}C))$ holds we need to solve $(\overline{i_1}-\underline{i_1})-(\overline{i_2}-\underline{i_2}) \geq 0$. Both upper bounds $\overline{i_1}, \overline{i_2}$ are calculated by selecting the greatest element from a set. The set of $\overline{i_2}$ is a subset of the set of $\overline{i_1}$ (see highlighted elements in Equations 3.9 and 3.10). Therefore $\overline{i_1}$ is at least as great a $\overline{i_2}$. Analogous $\underline{i_1}$ is at least a small as $\underline{i_2}$.

In order to find out whether $\overline{i_1}$ is greater or equal than $\overline{i_2}$ and whether $\underline{i_1}$ is less or equal than $\underline{i_2}$ we try to find cases, that fulfill the statements above.

We test systematically for all cases of closed intervals: Every closed interval can be assigned to one of the following six cases. Note: We exclude intervals which have no diameter (e.g. (a_1,a_1)). Intervals with no diameter behave like coefficients and are not subject to over-approximation.

Case	$A = [a_1; a_2]$
1	$a_1 < a_2 < 0$
2	$a_1 < a_2 = 0$
3	$a_1 < 0 < a_2$
4	$a_1 = 0 = a_2$
5	$a_1 = 0 < a_2$
6	$0 < a_1 < a_2$

Table 3.1: Possible assignments to set up an interval based on Definition 2.1.1.

We use this systematic approach to calculate both $AC+AB$ and $A(B+C)$ using every combination of cases as input for A, B and C. As input we use representative values that fulfill the according case. The results in Table 2 (appendix) show, that in about a fifth of all cases we can achieve a more narrow interval diameter and gain in precision when transforming the polynomial to a Horner scheme. None of the 216 cases resulted in a greater diameter which would mean a loss of precision.

Therefore we deduce that $d(A \cdot B + A \cdot C) \geq d(A(B + C))$ holds and thus we conclude that transforming a polynomial into a Horner scheme can reduce its over-approximation, but cannot increase it.

3.2.2 Effects on Precision Based on Interval Arithmetic

Next we compare the effects of interval arithmetics on precision: By comparing the definitions of multiplication from both arithmetics we can deduce a cause for the different results based solitary on the usage of different arithmetics: Let us take two intervals defined in midpoint-radius interval arithmetic (see Definition 2.1.9):

$$A = \langle \tilde{a}; \Delta_a \rangle, \quad B = \langle \tilde{b}; \Delta_b \rangle$$

Multiplying A and B in midpoint-radius interval arithmetic will give us:

$$a \cdot b = \langle \tilde{a}\tilde{b} \ ; \ |\tilde{a}|\Delta_b + |\tilde{b}|\Delta_a + \Delta_a\Delta_b \rangle$$

We now use the same intervals as above, but use standard interval arithmetic to multiply. $A \cdot B = C$ in standard interval arithmetic will give us the following interval: $C = [\underline{c}, \overline{c}]$ We solve the lower bound $\underline{c}$:

$$\underline{c} = min\{(\tilde{a}-\Delta_a)(\tilde{b}-\Delta_b),(\tilde{a}+\Delta_a)(\tilde{b}-\Delta_b),(\tilde{a}-\Delta_a)(\tilde{b}+\Delta_b),(\tilde{a}+\Delta_a)(\tilde{b}+\Delta_b)\}$$

We resolve to:

$$\begin{aligned} &ab + min\{-\Delta_a\tilde{b} + \Delta_a\Delta_b - \Delta_a\tilde{b}, \Delta_a\tilde{b} - \Delta_a\Delta_b - \Delta_a\tilde{a}, \\ &\qquad \Delta_a\tilde{b} - \Delta_a\Delta_b + \Delta_b\tilde{a}, \Delta_a\tilde{b} + \Delta_a\Delta_b + \Delta_b\tilde{a}\} \\ &= ab - max\{\Delta_a\tilde{b} - \Delta_a\Delta_b + \Delta_a\tilde{b}, -\Delta_a\tilde{b} + \Delta_a\Delta_b + \Delta_a\tilde{a}, \\ &\qquad -\Delta_a\tilde{b} + \Delta_a\Delta_b - \Delta_b\tilde{a}, -\Delta_a\tilde{b} - \Delta_a\Delta_b - \Delta_b\tilde{a}\} \end{aligned}$$

At this point we have to make an estimation:

$$\begin{aligned} &\leq ab - \Delta_a max\{-\tilde{b}, \tilde{b}\} - \Delta_a\Delta_b - \Delta_b max\{-\tilde{a}, \tilde{a}\} \\ &= ab - \Delta_a|\tilde{b}| - \Delta_a\Delta_b - \Delta_b|\tilde{a}| \end{aligned}$$

Analogous for the upper bound:

$$\bar{c} = max\{(\tilde{a} - \Delta_a)(\tilde{b} - \Delta_b),(\tilde{a} + \Delta_a)(\tilde{b} - \Delta_b),(\tilde{a} - \Delta_a)(\tilde{b} + \Delta_b),(\tilde{a} + \Delta_a)(\tilde{b} + \Delta_b)\}$$

We resolve to

$$\begin{aligned} &ab + max\{\Delta_a\tilde{b} - \Delta_a\Delta_b + \Delta_a\tilde{b}, -\Delta_a\tilde{b} + \Delta_a\Delta_b + \Delta_a\tilde{a}, \\ &\quad - \Delta_a\tilde{b} + \Delta_a\Delta_b - \Delta_b\tilde{a}, -\Delta_a\tilde{b} - \Delta_a\Delta_b - \Delta_b\tilde{a}\} \\ \geq &ab + \Delta_a max\{-\tilde{b}, \tilde{b}\} + \Delta_a\Delta_b + \Delta_b max\{-\tilde{a}, \tilde{a}\} \\ = &ab + \Delta_a|\tilde{b}| + \Delta_a\Delta_b + \Delta_b|\tilde{a}| \end{aligned}$$

We transform back into midpoint-radius interval arithmetic:

$$[ab - \Delta_a|\tilde{b}| + \Delta_a\Delta_b + \Delta_b|\tilde{a}| \ ; \ ab + \Delta_a|\tilde{b}| + \Delta_a\Delta_b + \Delta_b|\tilde{a}|] = \langle ab \ ; \ \Delta_a|\tilde{b}| + \Delta_a\Delta_b + \Delta_b|\tilde{a}|\rangle$$

We are able to deduce from standard interval arithmetic to midpoint-radius interval arithmetic, but not without estimating and therefore over-approximating our results. Thus, the gain in precision is related to some over-approximations which do not occur in standard interval arithmetic and therefore are not improvable by Horner schemes as postulated in the paper [7].
Using standard interval arithmetic and midpoint-radius interval arithmetic will in some cases generate different results (See Table 1).
So far our tests show that standard interval arithmetic provides more precise mathematical operations than midpoint-radius interval arithmetic. Thus we will not gain more precise results by switching the interval arithmetic we use in our implementation to midpoint-radius interval arithmetic.

3.3 Variable Selection Heuristics

After presenting the benefits of using Horner schemes, we present two approaches how to create a Horner scheme from a polynomial:

Selecting different variables to extract from a given multivariate polynomial will result in different Horner schemes and, as shown in Example 2.1.10 and 3.1.2, this can lead to different amount of arithmetic operations and different results. Therefore a heuristic to choose good candidates for variables to extract is required. Martine Ceberio and Vladik Kreinovich propose two strategies to chose variables [7]. In the following sections both strategies and their basic implementations will be presented.

Both strategy I and strategy II are variable selection heuristics that are based on structural features of the resulting Horner scheme. Both heuristics allow to precompute the Horner schemes once. This allows us to save computation time by reusing Horner schemes rather than creating new ones at every call of the constructor. Note, that there exist strategies, which depend in the current interval assignments of the variables and thus have to be recomputed.

3.3.1 Strategy I

Strategy I focuses on minimizing the amount of mathematical operations in the resulting Horner scheme. The procedure of the algorithm is as follows: First for each variable x_i within the polynomial $p(x_1, ...,x_n)$, we count those monomials N_i which contain the variable x_i and choose the variable with the highest occurrence. Note that we use a fixed variable ordering in case there are equal results. We now can represent the original polynomial as $p = p'_{dep} + p_{ind}$ where p_{ind} is the sum of all monomials that do not contain x_i and p'_{dep} is the sum of all monomials that do contain x_i. We transform to $p = p_{ind} + x_i \cdot p_{dep}$ where $p_{dep} = \frac{p'_{dep}}{x_i}$. Next we apply the algorithm for both p_{dep} and p_{ind} recursively.

Example 3.3.1 (Strategy I)**.** *Let us consider the following:*
$f(x_1,x_2,x_3) = x_1^2x_2 + x_1x_3 + x_1x_2x_3$ *For this polynomial* $N_1 = 3$,$N_2 = 2$ *and* $N_3 = 2$, $N_1 > N_2$,N_3 *therefore:*

$$f = A_0 + x_1A_1 \ \ where \ \ A_0 = 0, A_1 = x_1x_2 + x_3 + x_2x_3$$

Next we consider A_1: $N_1 = 1$, $N_2 = 2$ and $N_3 = 2$. N_2 and N_3 are greater than N_1 we chose N_2 because of lexicographical ordering, although it is irrelevant which one of the two we chose at this point.

$$A_1 = A'_0 + x_2 A'_1 \text{ where } A'_0 = x_3, A'_1 = x_1 + x_3$$
$$f = A_0 + x_1 A_1 = x_1(A'_0 + x_2 A'_1) = x_1(x_3 + x_2(x_1 + x_3))$$

Performance estimation: Based on preliminary testing we were able to get good results, but it is difficult to get a test set consisting of 'random' generated polynomials. The performance varies a lot depending on the properties of the input polynomials.
We recommend to test this further and to provide a context for which we can predict that this strategy will create good results. Compared to strategy II, which will be presented in the next section, this strategy takes less computation time. However, because this is a greedy algorithm, it might still not provide the optimal solution for the given polynomial. We present the results of both strategies in Chapter 5.

3.3.2 Strategy II

The second strategy is based on the second greedy algorithm presented in [7]. The strategy selects the variable which provides the highest reduction on the resulting interval diameter.

The algorithm operates as follows: Fist all variables from the input polynomial $p(x_1, ..., x_n)$ are gathered. Next we iterate over all variables $x_1 ... x_n$ within the polynomial. For each variable x_i we transform the polynomial to $p_i = p'_{dep} + p_{ind}$, where p'_{dep} represents the sum of all terms in which x_i occurs and p_{ind} represents the sum of all terms not containing x_i. Extract x_i from p'_{dep} yields $p_i = x_i \cdot p_{dep} + p_{ind}$. Next we replace all variables by their respective interval domains and solve the formula. The algorithm proceeds with the next variable until all variables where extracted at least once. The variable that caused the most narrow interval is selected for the creation of the Horner scheme. In case of equal results we use a fixed variable ordering. Next, the algorithm is applied recursively on both the polynomials p_{dep} and p_{ind}. At the end of the recursive process, the algorithm provides an order in which to extract the variables.

By mapping all variables to the same interval, we make the choice dependent on the impact a variable has on the result not based on its own value, but based on its coefficients and exponents.

Example 3.3.2. *Consider the polynomial $x_1^2 + 3x_2 + x_1 x_2$, with $x_1 = x_2 = [-2; 2]$. Strategy I yields the Horner scheme:*

$$x_1(x_1 + x_2) + 3x_2 = [-2; 2]([-2; 2] + [-2; 2]) + 3[-2; 2] = [-14; 14]$$

Strategy II yields the Horner scheme:

$$x_2(x_1 + 3) + x_1^2 = [-2; 2]([-2; 2] + 3) + [-2; 2]^2 = [-10{,}14]$$

Performance estimation: In our preliminary tests, strategy II created Horner schemes different from strategy I, but with similar amounts of arithmetic operations. The procedure of strategy II is a greedy algorithm and therefore does not guarantee that its result a Horner scheme with the highest precision possible. We expect strategy II to provide in average results with a higher precision than strategy I. At the same time strategy II needs more computation time than Strategy I, because it needs to evaluate more horner Horner schemes for each extraction candidate. The gain in precision might justify this trade-off.

Similar to strategy I, strategy II needs to be further tested on a representative set of polynomials, in order to be able to predict for which type of polynomials which strategy is optimal.

Chapter 4

Implementation

In the this section we present the implementation of the most important methods. First we present a constructor, that creates a Horner scheme based on a variable selection heuristic. Next we present a method called `simplify()`, which is an optional postprocessor, that reduces the wrapping effect by rearranging a Horner scheme. Finally we take a look at the evaluation method which is responsible for solving a Horner scheme.

We integrate the Horner schemes into the contraction procedure of the ICP module in order to speed up the evaluation process and increase the size of the contraction at the same time. The implementation of the propagation works mostly with linear formulas, which are not contactable by Horner schemes (see Section 3.1). Thus we will concentrate on improving the Newton's method (see Figure 2.7).

4.1 Constructor

The constructor for Horner schemes is split into two parts: a *recursive part* and a *non-recursive part*. The non-recursive part encapsulates the recursive part and manages the strategy. Based on the settings for a strategy a postprocessing in form of the `simplify()` method is executed. The recursive part is an constructor itself. This constructor is the part which actually is creating the data structure in its tree form, based on the strategy, as shown in Figure 4.1.

Listing 4.1: Constructor architecture

```
1 MultivariateHornerScheme(const Polynomial&){
2   //preprocessing
3
4   MultivariateHorenerScheme root(const Polynomial&,const intervalBox box);
5   this = root;
6
7   //postprocessing
8   if (strategy_requires_simplify){ simplify(); }
9 }
```

Next, we focus on the recursive part of the constructor: The creation of a Horner scheme is based on the variables which we extract. Therefore the Horner scheme is based on the order in which we select variables.

The recursive part of the constructor uses three parameters: the input polynomial p, the strategy s and the interval box B. We start by gathering all variables from p and analyzing the input polynomial based on the variable selection heuristic. Next the strategy suggests a variable x_i to extract. We continue by creating two new temporary polynomials: p_{dep} and p_{ind}. Every monomial containing x_i will be divided by x_i and added to p_{dep}, the monomials not containing the variable will be added to p_{ind}. Next we set the variable of our current Horner scheme instance to x_i. In case p_{dep} is not constant, we use it as input for the recursive constructor to create another Horner scheme. Then we save the result as the *dependent-part* of the current Horner scheme instance. In case p_{dep} is constant we will set the dependent-part as a `nullpointer` and save p_{dep} as the *dependent-coefficient* of the current Horner scheme instance. We save the *independent-part* and the *independent-coefficient* analogous.

In case the input polynomial is already a constant, the variable selection part is skipped and the polynomial will be directly set as *dependent-coefficient*.

Example 4.1.1 (Horner constructor)**.** *Let us consider the following term:*

$$4x^2y + 3x + 20z^3$$

We will create a Horner scheme by using strategy I. The result is:

$$x(x(y(4) + 3) + z(z(z(20)))$$

Figure 4.1 displays the resulting data structure of the recursive process.

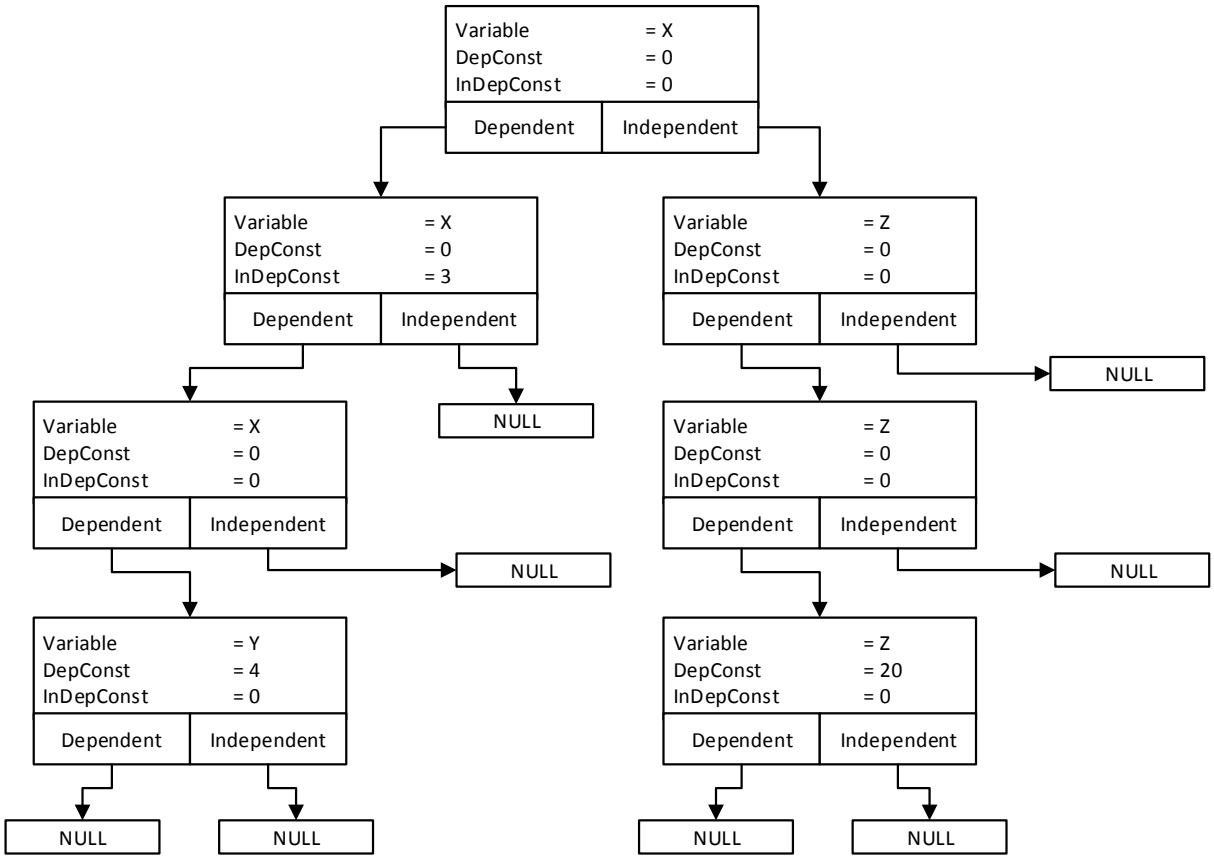


Figure 4.1: Diagram displaying the data structure of a Horner scheme of $4x^2y+3x^2+20z^3$ created with strategy I (3.3).

4.2 Simplification

The constructor currently (using strategy I or strategy II) only extracts single variables at a time. This can lead to chains of multiplications of variables and cause significant over-approximation (see Example 2.1.10). When creating a Horner scheme,

$$x^6 + y^4 \tag{4.1}$$

will be transformed to:

$$x(x(x(x(x(x))))) + y(y(y(y))) \tag{4.2}$$

Although it is impossible to increase the amount of calculations[1], that are needed to solve the input polynomial, we can loose information about it when transforming it into a Horner scheme. The input polynomial in our example (see Equation 4.1) contains the information that some multipliers are the same variable whereas in the

[1] Depending on the implementation of the power function it can make a difference, if values are calculated by multiplication or by power function. At this point we assume that $\prod_i x$ has the same computation cost as x^i.

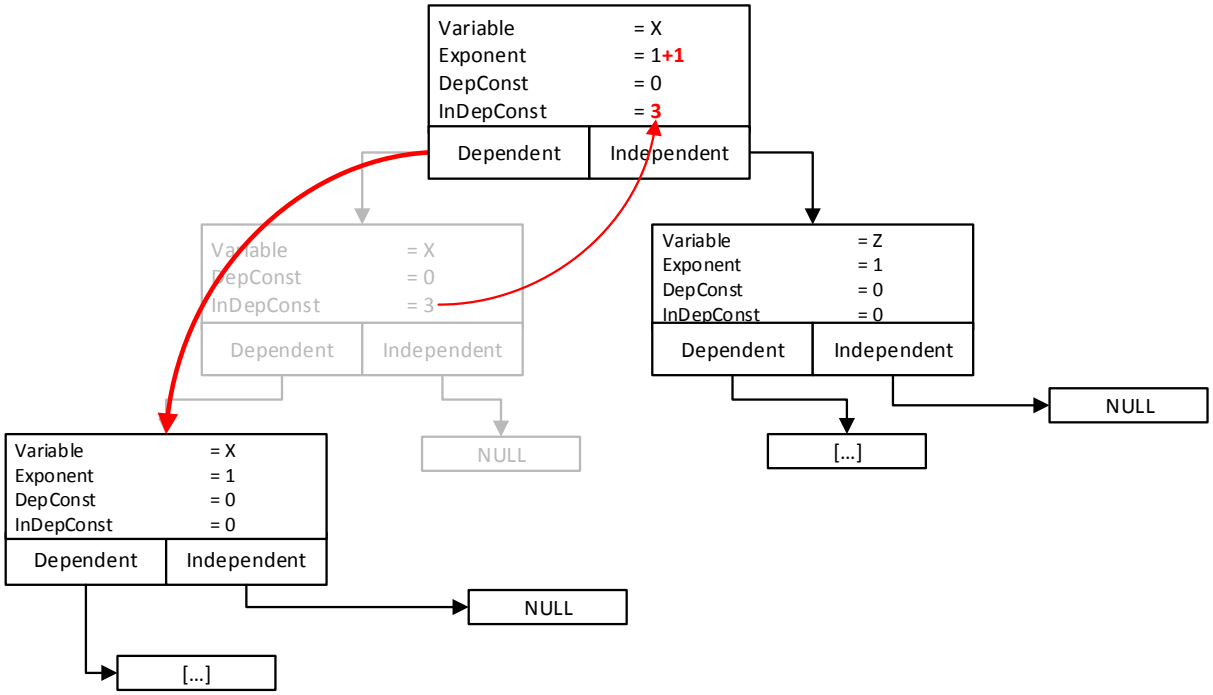


Figure 4.2: Diagram demonstrating the basic principle on which `simplify()` compresses the Horner scheme data structure.

Horner scheme (see Equation 4.2), we multiply variables where it is ambiguous whether they are the same or not.
Knowing that one multiplication of the same variables can be written as one variable being squared gives us the advantage to minimize the wrapping effect (see Section 2.4).
The `simplify()` method tries to make use of that advantage. Next to reducing over-approximation, `simplify()` also reduces the computation cost for the evaluation of the term. By reducing the overall amount of occurrences of variables in a polynomial, we reduce the amount of calls for the `evaluate()` method. The data structure of Horner schemes is based on the variables within the polynomial, `simplify()` compresses this data structure which makes it faster to traverse it.

Similar to the constructor this method works recursively. Each Horner scheme has a dependent part, and an independent part. Starting at the root h_{root} of the Horner scheme data structure it compares its own variable v_{root} to the variable v_{dep} of the dependent part $h_{root} \overset{dep}{\rightarrow} h_{dep}$. In case both are the same, it checks if the h_{dep} itself has no independent part $h_{dep} \overset{ind}{\rightarrow}$ `NULL`. If that is the case, the pointer to h_{dep} will be replaced with the pointer to the dependent part from h_{dep} ($h_{dep} \overset{dep}{\rightarrow} h'_{dep}$), so that $h_{root} \overset{dep}{\rightarrow} h'_{dep}$ and the exponent of the root is increased by one (see Figure 4.2).
The method continues with calling `simplify()` recursively with dependent and independent part one at a time as argument. The method only uses `simplify()` for a Horner scheme if the current pointer is not a `NULL`. Therefore this algorithm terminates as soon as the entire data structure is traversed.

4.3 Evaluation

We use the method `evaluate()` to solve the polynomials once we transformed them to Horner schemes. This method is implemented in a way that allows operator overloading in order to easily fit in with the preexisting code. The algorithm does a case distinction in order to traverse the data structure recursively In doing so the evaluate method gets called for every instance of every variable.

For evaluation we rely on the standard interval arithmetic provided by the framework. We do not provide new methods to evaluate the basic operations, because the gain in precision is reached by structural changes and not by changes in the arithmetic.

Example 4.3.1 (Evaluate)**.** *Given polynomial* $p = x_1(5 \cdot x_2 + 3x_3)$

$$\begin{aligned} evaluate(p) &= evaluate(x_1(5 \cdot x_2 + 3x_3)) \\ &= evaluate(x_1 \cdot evaluate(5 \cdot x_2 + 3x_3)) \\ &= evaluate(x_1 \cdot evaluate(evaluate(5 \cdot x_2) + evaluate(3 \cdot x_3))) \end{aligned}$$

The `evaluate()` method only traverses the data structure and uses the same basic arithmetic functions as the `evaluate()` method for multivariate polynomials. Due to the transformation, the amount of function calls that are needed for these basic arithmetic operations within the `evaluate()` method of the multivariate Horner scheme is lower on average, than in the `evaluate()` method of a multivariate polynomial.

Note that evaluate gets called for every node in our data structure (see Figure 4.1). Compressing the data structure, or choosing a more efficient variable selection heuristic will lead to fewer calls of the `evaluate()` method.

Chapter 5

Results

In this chapter we present and analyze the results of our implementation. First we introduce the parameter sets and conditions we used for the tests. Then we focus on the performance of the propagation method. Next we compare the performance of the Newton method without Horner schemes with the performance of the Newton method with the Horner schemes. We present a comparison for each strategy and observe the effect the `simplify()` method provides.

5.1 Benchmark Sets

In order to test and determine the performance of our implementation we use the same benchmark sets that are used to benchmark state-of-the-art SMT solvers, which are openly available [12], in SMTLIB 2 format.

- *hong* is a crafted dimension dependent benchmark set [10] and consists of 20 test cases.
- *zankl* is generated by termination analysis of term rewriting. Zankl contains 166 test cases.
- *keymaera* is generated by counterexample-guided synthesis [4] and consists of 421 test cases.
- *meti-tarski* is generated by theorem proving [5]. Meti-tarski is the greates test set we use. It contains 7713 test cases.

In order to get comparable test results we compile the same code for each setting with the same compiler. We used the same strategy to compose the same SMT solver with SMT-RAT (`RatTwo`). The benchmark tests are all tested on the same machine.

5.2 Propagation

The propagation method shows promising results (see Table 5.1). We use four test sets to test the method, *keymaera*, *zankl*, *hong* and *meti-tarski*. We test our implementation for each test set once with propagation activated and deactivated. Each test case has a limited time (200 seconds) to provide a result, once that time is exceeded we save the result as *TIMEOUT* and proceed to the next case. We log all test cases that solve in time as an *SAT* or *UNSAT*, and add up the required time. All *TIMEOUT* instances do not contribute to the overall solving time.

Table 5.1 displays a summary of the benchmark test for the propagation method. If we focus on the first three test sets (keymaera, zankl, hong) we observe a greater or at least equal rate of solved instances in both SAT and UNSAT. The overall solving time has decreased significantly. In the *hong* test set, using the propagation method nearly doubled the amount of test cases we can solve within the same time frame.

The fourth test set appears to be a contradiction to the assumption that the propagation method decreases the time needed to solve the test cases. Within the *meti-tarski* test set we could solve fewer problems and had more timeouts with propagation, than without. Analysing the test in detail shows, that in many instances the ICP module with propagation activated is faster than the module without, but at the same time there are a lot of test cases that require much more time than the implementation without propagation. In these cases the SMT solver spends most of the time in the solver module that succeeds the ICP module. Hence the ICP module could be more efficient, but this will not be visible in the test results, because the succeeding solver is using up most of the time. The strategy we use appoints VS (Virtual substitution) and CAD (Cylindrical

Test Set	SAT		UNSAT		TIMEOUT
	solved instances	solving time	solved instances	solving time	instances
keymaera (p)	0	0	402	**10.700**	19
keymaera	0	0	396	**19.038**	24
zankl (p)	30	**153.944**	16	639	115
zankl	28	**250.045**	16	690	117
hong (p)	0	0	**20**	**191**	**0**
hong	0	0	**11**	**473.866**	**9**
meti-tarski (p)	4759	3.991.758	2422	957.908	525
meti-tarski	4771	2.199.365	2430	934.495	519

Table 5.1: Comparison of performance of the ICP Module. The benchmark sets marked with (p) also used the propagation method to solve the test cases.

algebraic decomposition) as the succeeding solvers. It seems that in many cases the heuristic of CAD to find a good sampling point, simply failed and therefore took a lot of time to determine the solving process.

We can conclude that the propagation method improved the performance of the ICP module in many constraint-sets. Due to propagation more problems could be solved, and therefore the overall time to finish a test set was smaller.

5.3 Multivariate Horner with Strategy I

The goal behind strategy I is to provide a fast, yet efficient method to create a Horner scheme, that reduces the amount of overall mathematical operations within a polynomial. In Table 5.2a four benchmark tests for strategy I are listed. For each test set we compare the performance of the default setting of the SMT solver with the performance of the solver using Horner schemes and the solver using Horner schemes with the `simplify()` method. We log the outputs for each set of constraints within the benchmark test. In case the solver could determine the satisfiability we count the instance for either SAT or UNSAT, in both cases we log the time needed for solving as well. In case the solver took too much time to solve the constraint set (200 seconds), we will count that instance as a TIMEOUT and proceed to the next constraint set. Note, that in its default setting, the ICP module uses propagation.

5.3.1 Using Horner Schemes with Strategy I

Compared to the default setting of the SMT solver there was not one case where we solved more instances or where we saved some time by using the Horner scheme data structure. In the first test set *keymaera* there is one TIMEOUT instance less logged, this is due to a runtime error happening while solving one constraint set within the benchmark test. In the second test set *zankl* and in the third test set *hong* the usage of Horner schemes has no effect on the amount of solved instances. In average it takes more time to find the solutions when using Horner schemes, although the difference is below 10% in each case. In the last test set the usage of Horner schemes solved one constraint set less (SAT) than the default setting, the opposite of what we are trying to achieve. But by using Horner schemes we could solve two more constraint sets to UNSAT. Compared to the size of the *meti-tarski* benchmark these differences are negligible.

5.3.2 Using Simplify within Strategy I

Compared to the default setting, constructing Horner schemes with strategy I and a simplifyer does not seem to yield much improvements, but when comparing Horner schemes that were constructed without and with a simplifyer we notice some differences. Although we invest more computation cost to rearrange the Horner schemes, there seems to be a trend that simplified Horner schemes are faster to solve. In *keymaera* and *zankl* it took less time to solve the same amount of constraint sets when using the strategy combined with the simplify method. In *hong* it took more time solve the constraint sets after simplifying the Horner scheme. We can explain this by taking a look on the constraints within that test set. *Hong* contains rather simple polynomials[1]. It might not be possible to contract the data structure, but we still have to invest the computation cost to traverse it and will not save computation time by a compressed data structure. The *meti-tarski* test set displays

[1] Polynomials that have a small amount of terms in relation to a high amount of variables.

a shorter time needed to solve the constraint sets. Note that the solving time for the UNSAT cases in *meti-tarski* is not comparable, due to the fact, that different amount of constraints were solved.

5.4 Multivariate Horner with Strategy II

The goal of strategy II was to provide a Horner scheme that produces the smallest over-approximation possible. We expect this method to take more time because it needs to evaluate more potential Horner schemes. Similar to the benchmark results of strategy I, Table 5.2b displays the results of four benchmark test sets. We compare the performance of the SMT solver, with the modified version using the Horner scheme and with the Horner scheme combined with the simplify method.

Using Horner Schemes with Strategy II

Similar to strategy I we require more time to create the Horner schemes, than we save, once we use them (see Table 5.2b). In the first three test sets the same amount of problems are solved, but in every case the SMT solver using Horner schemes takes more time for solving, than the one not using Horner schemes. In the last test set (*meti-tarski*) we even solved fewer constraint sets than the SMT solver in its default set up.

Although we expect strategy II to require more time to form a Horner scheme, the strategy was faster than strategy I when solving constraint sets for the test set *zankl*. On the other hand it took longer when solving constraint sets in *hong* and *keymaera*. The results confirm our expectation, that the performance of a strategy is dependent on the type of the input set, and that there is not a strategy performing better on average.

5.4.1 Using Simplify within Strategy II

The purpose of the simplify method is to compress the Horner scheme and thereby save computation time. Table 5.2b shows that most constraint sets (except for the test set *hong*) took more time to be solved when using `simplify()` than solving it without the method. Preliminary tests have shown, that strategy II can tend to form Horner schemes which are not as easy compressible by the `simplify()` method as Horner schemes formed by strategy I. Strategy I tends to form multiple multiplications of the same variable in a row, that can be compressed to one variable with an exponent easily, thus explaining the difference. It seems that `simplify()` does not perform well in combination with strategy II, the benchmark set *hong* does not contain polynomials that get transformed a lot and therefore *hong* yields better results in combination with `simplify()`.

keymaera			
	D	Hs	$Hs\&S$
SAT	0	0	0
$\sum$ solving time	0	0	0
UNSAT	402	402	402
$\sum$ solving time	10.666	12.100	12.367
TIMEOUT	19	19	19

zankl			
	D	Hs	$Hs\&S$
SAT	30	30	30
$\sum$ solving time	154.725	153.262	157.863
UNSAT	16	16	16
$\sum$ solving time	659	680	715
TIMEOUT	115	115	115

hong			
	D	Hs	$Hs\&S$
SAT	0	0	0
$\sum$ solving time	0	0	0
UNSAT	20	20	20
$\sum$ solving time	203	635	611
TIMEOUT	0	0	0

meti-tarski			
	D	Hs	$Hs\&S$
SAT	4759	4758	4758
$\sum$ solving time	3.991.758	4.154.152	4.179.383
UNSAT	2428	2427	2427
$\sum$ solving time	957.908	998.863	901.028
TIMEOUT	525	527	527

(b) Test results for meti-tarski and strategy II

keymaera			
	D	Hs	$Hs\&S$
SAT	0	0	0
$\sum$ solving time	0	0	0
UNSAT	402	402	402
$\sum$ solving time	10.666	11.633	11.275
TIMEOUT	19	18	19

zankl			
	D	Hs	$Hs\&S$
SAT	30	30	30
$\sum$ solving time	154.725	158.063	156.710
UNSAT	16	16	16
$\sum$ solving time	659	686	679
TIMEOUT	115	115	115

hong			
	D	Hs	$Hs\&S$
SAT	0	0	0
$\sum$ solving time	0	0	0
UNSAT	20	20	20
$\sum$ solving time	203	227	301
TIMEOUT	0	0	0

meti-tarski			
	D	Hs	$Hs\&S$
SAT	4759	4758	4758
$\sum$ solving time	3.991.758	4.226.384	4.026.107
UNSAT	2428	2430	2429
$\sum$ solving time	957.908	934.495	916.182
TIMEOUT	525	524	525

(a) Test results for meti-tarski and strategy I

Table 5.2: Summary of the benchmark results. D = Defauls set up, Hs = Horner scheme that were not further simplified, $Hs\&S$ = Horner scheme with the usage of `simplify()`

5.5 Reduction of Contractions within the ICP Module

As presented in Section 5.3 and Section 5.4 the usage of the Horner scheme does not improve the performance of the ICP module as expected. In order to quantify the gain in precision by Horner schemes (strategy I with simplification), we logged the amount of contractions performed within the ICP module. For this test we deactivated the propagation.

Keymaera contains 412 test cases. At least 106 of these test cases are contracted by the ICP module, the rest can be sorted out previously by the SAT solver. Note that 25 cases needed to be stopped by a timeout. When using Horner schemes 6 of the previously contracted test cases need less contractions than before. The amounts of contractions are displayed in Table 5.3.

Testing the benchmark set *zankl* for contraction yielded 40 of 166 cases that contracted the remaining cases needed to be stopped because they took to much time (timeout). 7 of the 40 cases required less contractions than before. The counts of contractions are displayed in Table 5.4.

keymaera		
Result	Contractions without Horner schemes	Contractions with Horner schemes
unsat	7	5
unsat	1607	1038
unsat	30	29
unsat	1607	1038
unsat	27	22
unsat	90	81

Table 5.3: Reduction of contractions, due to the usage Horner schemes

zankl		
Result	Contractions without Horner schemes	Contractions with Horner schemes
sat	5489	5132
sat	3427	3422
sat	**4560**	**3131**
sat	1120	1108
sat	1705	1596
unsat	122	118
sat	5098	5069

Table 5.4: Reduction of contractions, due to the usage Horner schemes

The amount of the reduction of contractions depends a lot on the in the input constraints. In our test sets it varies from less that 1% to 46%.

Chapter 6

Conclusion

In this final chapter we present our findings, and point out areas for further improvement.

6.1 Future Work

The presented Horner scheme data structure still provides room for improvement. Horner schemes as presented in this thesis provide a lot of useful properties, which could be extended to other operations within the ICP module.

6.1.1 Improving the Variable Selection Heuristics

The variable selection heuristic has a lot of influence on the resulting Horner scheme and the time needed to create it. In this thesis we only present two strategies, further strategies might yield better results. Currently the strategy is defined by the user. A decision process could be implemented, that chooses a strategy based on the input polynomial (e.g. the ratio of terms to variables).

A concept for another variable selection heuristic would be strategy III. The idea behind strategy III is similar to the principle strategy II is based on. The variable which has the highest impact on the precision is selected. This will be achieved by replacing the variables with their respective domains from the interval box and solving the polynomial with each variable extracted one at a time. Instead of replacing the interval box with a default value for each variable as in strategy II, all variables will be updated after each iteration of the contractor. This might cause the constructor to create a new Horner scheme at every iteration, but at the same time might lead to a more precise result. In order to manage a trade off, a counter could be implemented allowing a recomputation of a Horner scheme with strategy III every n contractions.

It would need to be tested if the gain in precision, and therefore the fewer contractions, would trade off the increased amount of computing time for horner scheme constructions.

6.1.2 Multivariate Polynomial to Univarate Polynomial Transformation

All over-approximations related to the wrapping effect are caused by the fact that the evaluating method is not checking whether two variables are identical or not. Instead of adding this feature to multiplication and subtraction, multivariate polynomials could be transformed to univariate polynomials before evaluation. The univariate polynomial would represent the multivariate polynomial from the 'perspective' of one variable. Consequently all other variables would be replaced by their values and function as coefficients.

The evaluation of univariate polynomials is not affected by the wrapping effect, because in case of a multiplication or a subtraction of two variables there is ambiguity whether both variables are the same, and according actions can be taken to reach the correct result.

6.1.3 Horner Schemes in Preprocessing

This thesis mainly tries to exploit the benefits of precision and efficiency that come with Horner schemes, but they are also very useful when searching for zero points of polynomials. The following polynomial can be split into factors, which permits an alternative way to transform sets of constraints.

$$\begin{aligned} 2x_1 + 3x_1x_2 + 5x_1 = 0 \Leftrightarrow x_1 \cdot (2 + 3x_2 + 5) = 0 \\ x_1 = 0 \vee 7 + 3x_2 = 0 \end{aligned}$$

A new set of constraints, based on the factorization of the polynomial could be added to the original set of constraints. In the best case the satisfiability can be determined immediately after extracting one key variable from all monomials. In the worst case the polynomial can not be efficiently factorized and therefore we will not have simpler constraints. The result will not add any new constraints while still needing a lot of computation time.

The challenge when realizing this preprocessor among others is the development of an efficient variable selection heuristic for the multivariate Horner schemes. Both strategies as they are presented in this thesis might not be useful, because both are focused on the resulting interval diameter or rather the efficiency and not on the satisfiability of the constraint.

6.2 Conclusion

In this thesis we were able to prove, that in theory using Horner schemes can provide not only a more efficient way to evaluate polynomials, but also decreases the over-approximation while doing so. The benchmark tests we used to evaluate our implementation show, that both strategy I and strategy II do not increase the overall efficiency of the ICP module but rather lower it, spending more time for the average solving process. We tested on a wide variety of different constraint sets and could not find a test set that provides the expected positive results. There are many obstacles that prevent the efficient application of Horner schemes, such as the preprocessing, input polynomials that cannot be contracted much further and the deployment of a variable selection heuristic that performs well, for any kind of polynomial.

Analyzing the contraction process in more detail, showed us, that the amount of contractions can be reduced by Horner schemes, but this effect is negligible when we use Horner schemes combined with propagation. Once we use propagation the implemented data structure does not have any positive effect on the efficiency. In order for the Horner scheme data structure to work, we need to make up the time, we invest in creating the Horner schemes. In all tested cases, this trade-off does not seem to work out. It is desirable to test Horner schemes with a test set, that consists of polynomials that can be contracted a lot. We expect Horner schemes to work better with polynomials, whose number of terms is far greater than the number of contained variables.

The interval propagation method which was implemented as part of this thesis, shows promising results and proves to increase the overall efficiency of the ICP solver. It provides the ICP solver with another contraction method, that is cheap in computation time, but still reduces the average search space significantly.

We were able to increase the efficiency by implementing a new contraction method, and we were able to increase the precision, but sadly consequently lowered the efficiency.

Bibliography

[1] The complexity of linear problems in fields. *Journal of Symbolic Computation*, 5(1 - 2):3 – 27, 1988.

[2] *Gröbner Bases: A Computational Approach to Commutative Algebra*. Springer-Verlag New York, 1993.

[3] Florian Corzelius , Ulrich Loup , Sebastian Junges , Erika Abraham. *SMT-RAT: An SMT-comliant nonlinear real arithmetic toolbox.* 2012.

[4] Platzer, Andre, Quesel, Jan-David, and Ruemmer, Philipp. Real world verification. In *Automated Deuduction - CADE-22*, volume 5663 of *Lecture Notes in Computer Science*, pages 485–501. Springer Berlin Heidelberg, 2009.

[5] Akbarpour, Behzad and Paulson, Lawrence Charles. Meti Tarski: An automatic theorem prover for real-valued special functions. *Journal of Automated Reasoning*, 44(3):175–205, 2010.

[6] Christopher W. Brown and James H. Davenport. The complexity of quantifier elimination and cylindrical algebraic decomposition. In *Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation*, ISSAC '07, pages 54–60. ACM, 2007.

[7] Martine Ceberio and Vladik Kreinovich. Greedy algorithms for optimizing multivariate horner schemes. *SIGSAM Bull.*, 38(1):8–15, March 2004.

[8] George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decompostion. In *Automata Theory and Formal Languages 2nd GI Conference Kaiserslautern, May 20 - 23, 1975*, volume 33 of *Lecture Notes in Computer Science*, pages 134–183. Springer Berlin Heidelberg, 1975.

[9] Florian Corzillus. Manual SMT-RAT 2.0.0. "`https://github.com/smtrat/smtrat/wiki`", 2015. [Online; accessed 24th September 2015].

[10] Hoon Hong. Comparison of several decision algorithms for the existential theory of the reals. Technical report, 1991.

[11] W. G. Horner. A new mothod of solving numerical equations of all orders, by continuous approximation. *Philosophical Transactions of the Royal Society of London*, 1819.

[12] The SMT-LIB Initiative. SMT-LIB. "`http://smtlib.cs.uiowa.edu/`", 2015. [Online; accessed 24th September 2015].

[13] Kulisch, Ulrich W. Complete interval arithmetic and its implementation on the computer. In *Numerical Validation in Current Hardware Architectures*, volume 5492 of *Lecture Notes in Computer Science*, pages 7–26. Springer Berlin Heidelberg, 2009.

[14] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, 2009.

[15] Siegfried M. Rump. Fast and parallel interval arithmetic. *BIT Numerical Mathematics*, 39(3):534–554, 1999.

[16] Stefan Schupp. Interval constraint propagation in smt compliant decision procedures. Master's thesis, RWTH Aachen, 2013.

[17] Franjo Ivancic Aarti Gupta Sriram Sankaranarayanan Edmund M. Clarke Sicun Gao, Malay Ganai. Integrating ICP and LRA Solvers for Deciding Nonlinear Real Arithmetic Problems. *Procedings of Formal Methods in Computer-Aided Desing (FM-CAD'10)*, 2010.

[18] Dahmen, W. and Reusken, A. *Numerik für Ingenieure und Naturwissenschaftler*. Springer-Lehrbuch. Springer Berlin Heidelberg, 2008.

Appendix

A	B	$A \cdot B$ in midpont-radius interval arithmetic	$A \cdot B$ in interval arithmetic as used in our implementation	Diameter comparison
$[-200, -100]$	$[-200, -100]$	$[10000, 40000]$	$[5000, 40000]$	GREATER
$[-200, -100]$	$[-100, 0]$	$[-0, 20000]$	$[-5000, 20000]$	GREATER
$[-200, -100]$	$[-100, 100]$	$[-20000, 20000]$	$[-20000, 20000]$	SAME
$[-200, -100]$	$[0, 0]$	$[0, 0]$	$[-0, 0]$	SAME
$[-200, -100]$	$[0, 100]$	$[-20000, -0]$	$[-20000, 5000]$	GREATER
$[-200, -100]$	$[100, 200]$	$[-40000, -10000]$	$[-40000, -5000]$	GREATER
$[-100, 0]$	$[-200, -100]$	$[-0, 20000]$	$[-5000, 20000]$	GREATER
$[-100, 0]$	$[-100, 0]$	$[0, 10000]$	$[-5000, 10000]$	GREATER
$[-100, 0]$	$[-100, 100]$	$[-10000, 10000]$	$[-10000, 10000]$	SAME
$[-100, 0]$	$[0, 0]$	$[0, 0]$	$[-0, 0]$	SAME
$[-100, 0]$	$[0, 100]$	$[-10000, 0]$	$[-10000, 5000]$	GREATER
$[-100, 0]$	$[100, 200]$	$[-20000, 0]$	$[-20000, 5000]$	GREATER
$[-100, 100]$	$[-200, -100]$	$[-20000, 20000]$	$[-20000, 20000]$	SAME
$[-100, 100]$	$[-100, 0]$	$[-10000, 10000]$	$[-10000, 10000]$	SAME
$[-100, 100]$	$[-100, 100]$	$[-10000, 10000]$	$[-10000, 10000]$	SAME
$[-100, 100]$	$[0, 0]$	$[0, 0]$	$[0, 0]$	SAME
$[-100, 100]$	$[0, 100]$	$[-10000, 10000]$	$[-10000, 10000]$	SAME
$[-100, 100]$	$[100, 200]$	$[-20000, 20000]$	$[-20000, 20000]$	SAME
$[0, 0]$	$[-200, -100]$	$[0, 0]$	$[-0, 0]$	SAME
$[0, 0]$	$[-100, 0]$	$[0, 0]$	$[-0, 0]$	SAME
$[0, 0]$	$[-100, 100]$	$[0, 0]$	$[0, 0]$	SAME
$[0, 0]$	$[0, 0]$	$[0, 0]$	$[0, 0]$	SAME
$[0, 0]$	$[0, 100]$	$[0, 0]$	$[0, 0]$	SAME
$[0, 0]$	$[100, 200]$	$[0, 0]$	$[0, 0]$	SAME
$[0, 100]$	$[-200, -100]$	$[-20000, -0]$	$[-20000, 5000]$	GREATER
$[0, 100]$	$[-100, 0]$	$[-10000, 0]$	$[-10000, 5000]$	GREATER
$[0, 100]$	$[-100, 100]$	$[-10000, 10000]$	$[-10000, 10000]$	SAME
$[0, 100]$	$[0, 0]$	$[0, 0]$	$[0, 0]$	SAME
$[0, 100]$	$[0, 100]$	$[0, 10000]$	$[-5000, 10000]$	GREATER
$[0, 100]$	$[100, 200]$	$[0, 20000]$	$[-5000, 20000]$	GREATER
$[100, 200]$	$[-200, -100]$	$[-40000, -10000]$	$[-40000, -5000]$	GREATER
$[100, 200]$	$[-100, 0]$	$[-20000, 0]$	$[-20000, 5000]$	GREATER
$[100, 200]$	$[-100, 100]$	$[-20000, 20000]$	$[-20000, 20000]$	SAME
$[100, 200]$	$[0, 0]$	$[0, 0]$	$[0, 0]$	SAME
$[100, 200]$	$[0, 100]$	$[0, 20000]$	$[-5000, 20000]$	GREATER
$[100, 200]$	$[100, 200]$	$[10000, 40000]$	$[5000, 40000]$	GREATER

Table 1: Systematic comparison of interval multiplications

Case	$A = [a_1,a_2]$	$B = [b_1,b,2]$	$C = [c_1,c_2]$
1	$a_1 < a_2 < 0$	$b_1 < b_2 < 0$	$c_1 < 0 < c_2$
2	$a_1 < a_2 < 0$	$b_1 < b_2 < 0$	$c_1 = 0 < c_2$
3	$a_1 < a_2 < 0$	$b_1 < b_2 < 0$	$0 < c_1 < c_2$
4	$a_1 < a_2 < 0$	$b_1 < b_2 = 0$	$0 < c_1 < c_2$
5	$a_1 < a_2 < 0$	$b_1 < 0 < b_2$	$c_1 < c_2 < 0$
6	$a_1 < a_2 < 0$	$b_1 < 0 < b_2$	$0 < c_1 < c_2$
7	$a_1 < a_2 < 0$	$b_1 = 0 < b_2$	$c_1 < c_2 < 0$
8	$a_1 < a_2 < 0$	$0 < b_1 < b_2$	$c_1 < c_2 < 0$
9	$a_1 < a_2 < 0$	$0 < b_1 < b_2$	$c_1 < c_2 = 0$
10	$a_1 < a_2 < 0$	$0 < b_1 < b_2$	$c_1 < 0 < c_2$
11	$a_1 < a_2 = 0$	$b_1 < b_2 < 0$	$c_1 < 0 < c_2$
12	$a_1 < a_2 = 0$	$b_1 < b_2 < 0$	$c_1 = 0 < c_2$
13	$a_1 < a_2 = 0$	$b_1 < b_2 < 0$	$0 < c_1 < c_2$
14	$a_1 < a_2 = 0$	$b_1 < b_2 = 0$	$0 < c_1 < c_2$
15	$a_1 < a_2 = 0$	$b_1 < 0 < b_2$	$c_1 < c_2 < 0$
16	$a_1 < a_2 = 0$	$b_1 < 0 < b_2$	$0 < c_1 < c_2$
17	$a_1 < a_2 = 0$	$b_1 = 0 < b_2$	$c_1 < c_2 < 0$
18	$a_1 < a_2 = 0$	$0 < b_1 < b_2$	$c_1 < c_2 < 0$
19	$a_1 < a_2 = 0$	$0 < b_1 < b_2$	$c_1 < c_2 = 0$
20	$a_1 < a_2 = 0$	$0 < b_1 < b_2$	$c_1 < 0 < c_2$
21	$a_1 < 0 < a_2$	$b_1 < b_2 < 0$	$c_1 = 0 < c_2$
22	$a_1 < 0 < a_2$	$b_1 < b_2 < 0$	$0 < c_1 < c_2$
23	$a_1 < 0 < a_2$	$b_1 < b_2 = 0$	$c_1 = 0 < c_2$
24	$a_1 < 0 < a_2$	$b_1 < b_2 = 0$	$0 < c_1 < c_2$
25	$a_1 < 0 < a_2$	$b_1 = 0 < b_2$	$c_1 < c_2 < 0$
26	$a_1 < 0 < a_2$	$b_1 = 0 < b_2$	$c_1 < c_2 = 0$
27	$a_1 < 0 < a_2$	$0 < b_1 < b_2$	$c_1 < c_2 < 0$
28	$a_1 < 0 < a_2$	$0 < b_1 < b_2$	$c_1 < c_2 = 0$
29	$a_1 = 0 < a_2$	$b_1 < b_2 < 0$	$c_1 < 0 < c_2$
30	$a_1 = 0 < a_2$	$b_1 < b_2 < 0$	$c_1 = 0 < c_2$
31	$a_1 = 0 < a_2$	$b_1 < b_2 < 0$	$0 < c_1 < c_2$
32	$a_1 = 0 < a_2$	$b_1 < b_2 = 0$	$0 < c_1 < c_2$
33	$a_1 = 0 < a_2$	$b_1 < 0 < b_2$	$c_1 < c_2 < 0$
34	$a_1 = 0 < a_2$	$b_1 < 0 < b_2$	$0 < c_1 < c_2$
35	$a_1 = 0 < a_2$	$b_1 = 0 < b_2$	$c_1 < c_2 < 0$
36	$a_1 = 0 < a_2$	$0 < b_1 < b_2$	$c_1 < c_2 < 0$
37	$a_1 = 0 < a_2$	$0 < b_1 < b_2$	$c_1 < c_2 = 0$
38	$a_1 = 0 < a_2$	$0 < b_1 < b_2$	$c_1 < 0 < c_2$
39	$0 < a_1 < a_2$	$b_1 < b_2 < 0$	$c_1 < 0 < c_2$
40	$0 < a_1 < a_2$	$b_1 < b_2 < 0$	$c_1 = 0 < c_2$
41	$0 < a_1 < a_2$	$b_1 < b_2 < 0$	$0 < c_1 < c_2$
42	$0 < a_1 < a_2$	$b_1 < b_2 = 0$	$0 < c_1 < c_2$
43	$0 < a_1 < a_2$	$b_1 < 0 < b_2$	$c_1 < c_2 < 0$
44	$0 < a_1 < a_2$	$b_1 < 0 < b_2$	$0 < c_1 < c_2$
45	$0 < a_1 < a_2$	$b_1 = 0 < b_2$	$c_1 < c_2 < 0$
46	$0 < a_1 < a_2$	$0 < b_1 < b_2$	$c_1 < c_2 < 0$
47	$0 < a_1 < a_2$	$0 < b_1 < b_2$	$c_1 < c_2 = 0$
48	$0 < a_1 < a_2$	$0 < b_1 < b_2$	$c_1 < 0 < c_2$

Table 2: 48 Cases of 216 in which $A \cdot (B + C)$ resulted in a more narrow interval than $AB + AC$. All remaining 168 Cases resulted in the equal intervals. There was no case in which $A \cdot (B + C)$ solved to a greater interval than $AB + AC$.