UNIVERSITÄT PASSAU

Fakultät für Informatik und Mathematik

Master thesis

Improving the Efficiency of Code Generation Based on Cylindrical Algebraic Decomposition

Thomas Lang

Supervisor:	Prof. Christian Lengauer, Ph.D.
Second reader:	Prof. Dr. Martin Griebl
Tutor:	Dr. Armin Größlinger

21st March 2018

Abstract

Ever since the introduction of the Polyhedron model for loop optimization, generalizations of the model to make it applicable to a bigger class of programs have been developed. Prime examples for this include the generalization to handle non-perfectly nested loop nests or permitting WHILE loops. However, a main restriction that remained was that all loop bounds and array subscripts had to be linear in both the iterator variables and the runtime parameters. Previous work also showed that this restriction can be removed and that polynomial constraints can be handled in theory. The adoption of this extension to the model has been hindered by the computational expensiveness of the underlying algorithm, cylindrical algebraic decomposition (CAD).

This thesis focusses on the code generation part of this generalization and presents an implementation in C++ which incorporates several improvements for cases relevant in practice.

In the first part, we give a general overview about the problem of code generation for polynomial constraints and describe how they can be overcome using CAD. This is followed by a description of our implementation of this method in C++ and several optimizations. The optimizations include both algorithmic enhancements and tuning of the code generation for important special cases which occur in practice. The third part of this thesis compares the code generation speed of our implementation to an existing prototype of this method written in Haskell. Finally, we discuss possible future work that could improve the performance of our code generation procedure even more.

Acknowledgements
This thesis has only been made possibly through the support of many people.

First of all, I want to thank Prof. Christian Lengauer, Ph.D., who gave me the opportunity of writing my master thesis in the field of code generation and code optimization.

Next, I want to thank my tutor Dr. Armin Größlinger, who worked out the theoretical background of this thesis in his dissertation and provided me hints about some optimizations as well as runtime comparisons of his prototypical implementation. He was a valuable dialogue partner who always offered me insight in optimization techniques and the mathematical background of this thesis.

Lastly, I appreciate the remaining team of the chair of programming for giving me various hints on tools that supported me during the development of this work.

Contents

List of Figures

List of Algorithms

Listings

List of Tables

Chapter 1

Introduction

Decades years ago, scientists started to express automated code optimization problems like parallelization in a mathematically sound model known as the *Polytope model*. This model had several restrictions like that all loops had to be perfectly nested or that `WHILE` loops were not permitted due to the boundedness of polytopes. But the most obvious restriction was that all recurrence equations (i.e. equations on array index functions) had to be affine expressions in loop iterators using only constant factors and effectively constant offset [Len93].

Since its beginnings much work was invested in order to generalize the Polytope model as far as possible. For example, the model was generalized to allow imperfectly nested loops [Len93] and even `WHILE` loops can be handled now [Gri96], giving the model a new name of *Polyhedron model*.

However, the restriction of affine expressions remained for a long time, until [Grö09] showed that all phases of polyhedral optimization can be generalized to allow arbitrary polynomial expressions.

Naturally, this generalization makes all phases more complicated, especially the phase of code generation on which we focus.

Firstly, we describe the problems of code generation in this generalized context and how these problems can be solved theoretically using the method of Cylindrical Algebraic Decomposition.

Then we will provide a description of our implementation of such a code generation in the generalized polyhedron model in `C++` based on the freely available library *SMT-RAT* [1].

This chapter also includes a detailed description of several optimizations we applied to this basic implementation in order to enhance its performance. These optimizations include both algorithmic improvements applicable to every input problem, as well as optimizations tuned specifically for handling certain special cases which occur often in practice.

In Chapter 4, we evaluate the performance of our implementation by comparing our basic implementation with an existing prototype in `Haskell`. Furthermore, we compare our basic implementation against itself with optimizations applied one after another.

Finally, we conclude our work by stating how we improved the performance of the described more general code generation and what further improvements might speed it up even more.

[1] `https://github.com/smtrat/smtrat`

Chapter 2

Prerequisites

In this first chapter, we want to introduce the reader to the concepts of code generation as well as to the problems occuring if non-linearities are introduced.

2.1. The Code Generation Problem

As stated, the Polyhedron model takes a source program and transforms it into a model-based representation, on which several optimizations like parallelization, vectorization or cache locality optimization are performed, resulting in a transformed model.
But clearly, such a model is not executable. Therefore, we need to output the transformed model as source code which can be compiled and executed afterwards.

2.1.1. Basic Code Generation

Before we step into code generation, let us define some basic terminology.

Definition 2.1 (Domain). *A set $D \subset \mathbb{Z}^n, n \in \mathbb{N}$ which enumerates all iteration points of a program, is called a* Domain *or* Index set.

Definition 2.2 (Statement). *A* Statement *is a set of instructions which are executed at each point of a domain. We will abbreviate statements with T_i where $i \in \mathbb{N}$ marks the domain to which T belongs. If T depends on loop iterators $x_1, \ldots, x_n$ we write $T(x_1, \ldots, x_n)$.*

In practice, statements often depend on each other. Other transformations in the polyhedron model manipulate the structure of the program to become more efficient while maintaining dependencies. We will not cover these optimizations here as they can be found in related literature, but we need to respect the *lexicographic order* which plays a key role in the stated optimizations.

Definition 2.3. *Let $n \in \mathbb{N}$ and let $i, j \in \mathbb{Z}^n$ be two vectors representing the loop iterators in certain iterations. We define the lexicographic order $\prec_{lex}$ recursively as follows:*

$$() \prec_{lex} ()$$

$$(i_1, i_2, \ldots, i_n) \prec_{lex} (j_1, j_2, \ldots, j_n) \Leftrightarrow (i_1 < j_1) \vee ((i_1 = j_1) \wedge (i_2, \ldots, i_n) \prec_{lex} (j_2, \ldots, j_n))$$

As the name *Polyhedron model* suggests, many iteration domains can be expressed mathematically as a polyhedron or as finite union of polyhedra.

Definition 2.4 (Polyhedron). *Let $M \in \mathbb{R}^{m \times n}$ be a matrix and let $q \in \mathbb{Z}^m$ denote an offset vector. Then the set*

$$P = \{x \in \mathbb{Z}^n \mid Mx \geq q\}$$

is called a Polyhedron.

We see that such a polyhedron consists of expressions involving polynomials. Polynomials are commonly defined as follows:

Definition 2.5 (Polynomial)**.** *Let $\mathbb{K} \in \{\mathbb{Q}, \mathbb{R}\}$ be a field and let $d \in \mathbb{N}$.*
A function is called a n-variate polynomial over $\mathbb{K}$ if and only if it is of the form

$$\begin{aligned} p\colon \mathbb{K}^n &\to \mathbb{K} \\ (x_1, \dots, x_n)^T &\mapsto \sum_{j=1}^{d} a_j x^{\alpha_j} \end{aligned}$$

with $x^{\alpha_j} = x_1^{\alpha_{j1}} x_2^{\alpha_{j2}} \cdots x_n^{\alpha_{jn}}$ being a multi-index expression where $\alpha_{11}, \dots, \alpha_{dn} \in \mathbb{N}_0$.
The set of all n-variate polynomials with rational coefficients is denoted by $\mathbb{Q}[x_1, \dots, x_n]$.
Throughout this thesis, we will also consider n-variate polynomials as univariate polynomials whose coefficients are multivariate polynomials i.e. $\mathbb{Q}[x_1, \dots, x_n] = (\mathbb{Q}[x_1, \dots, x_{n-1}])[x_n]$.

The polynomials acting as constraints encompassing the polyhedral domain are so-called *affine-linear* polynomials, which are a special subclass of the set of polynomials.

Definition 2.6. *A polynomial $p \in \mathbb{Q}[x_1, \dots, x_n]$ is called* affine-linear *if it has the shape*

$$p = \alpha x_n + \beta,$$

where $\alpha, \beta \in \mathbb{Q}[x_1, \dots, x_{n-1}]$.
Such a polynomial is called linear *if $\beta = 0$.*
In this thesis, we will use the terms affine-linear, affine *and* linear *interchangeably.*

Visually, such a polyhedron is a description of a set of polynomials which encompass the points of execution. As a speciality, such a polyhedron does not need to be bounded in all directions.

In the more general context of our code generation we can lift the idea of contraints defined by polynomials to any dimension.

Definition 2.7. *Let $p \in \mathbb{Q}[x_1, \dots, x_n]$ be a polynomial. We call an (in-)equality of the form*

$$p \sim 0, \qquad \sim \in \{<, \leq, =, \neq, \geq, >\}$$

a polynomial constraint.

Let us now consider an example of two domains which are overlapping in a certain area. Consider the domains $D_1 = \{x \in \mathbb{Z} \mid 0 \leq x \leq 4\}$ and $D_2 = \{x \in \mathbb{Z} \mid p \leq x \leq 2\}$ with $p \in \mathbb{Z}$ being a runtime parameter and let T_i be some statement executed in domain i, $i = 1, 2$.

If we want to generate code for these two domains, we must take the lexicographic order on x into account i.e. we must not produce independent loops like

```
for (x = 0; x <= 4; ++x)
  T1
for (x = p; x <= 2; ++x)
  T2
```

Listing 2.1: Lexicographically incorrect code for two overlapping iteration domains

That is because for $x = 2$ (if $p = 2$), both T_1 and T_2 have to be executed in the same iteration in the model, but not after all iterations of the first loop as shown in the code. This would be easily decidable if the value for p was known at compile-time, but since p is a runtime parameter, the generated code has to perform correctly for *any* possible integral value p can take.

Such complications when introducing runtime parameters make code generation a hard problem. But as already [QRW00, pg. 478] pointed out, this problem can be solved by considering the union of all domains active (in our example D_1 and D_2) and partitioning this union into disjoint subsets. Then, for each such subset an individual loop is generated.

The perhaps easiest way of doing so is by performing a case distinction on the values of p and generate loop nests enumerating the respective points and statements.

Applied to the above example, this would yield

```
if (p <= 0) {
  for (x = p; x <  0; ++x) { T2;      }
  for (x = 0; x <= 2; ++x) { T1;T2;   }
  for (x = 3; x <= 4; ++x) { T1;      }
} else if (p > 0 && p <= 2) {
  for (x = 0; x <  p; ++x) { T1;      }
  for (x = p; x <= 2; ++x) { T1;T2;   }
  for (x = 3; x <= 4; ++x) { T1;      }
} else {
  for (x = 0; x <= 4; ++x) { T1;      }
}
```

Listing 2.2: Decomposition of a one-dimensional index space

As a remark please note that in this case distinction the `else` branch does not have a loop executing the statement T_2 as for $p > 2$ the iteration domain is empty. This speciality can be used in this generation by not generating that loop, whereas in common polyhedral code generation this would not be done as the *emptiness* of a loop nest cannot be decided for such parameterized loops in general.

2.1.2. Code Generation for Non-linear domains

Slightly more complicated is the code generation for the domain [1] $D_3 = \{(x, y) \in \mathbb{Z}^2 | 1 \leq x \leq 7 \wedge 1 \leq y \leq 9 \wedge (y-4)^2 - 3x + 12 \geq 0\}$, which clearly contains non-linearities. This domain cannot be treated by common polyhedral code generation. Figure 2.1 on the next page shows this domain graphically and Listing 2.3 on the following page shows the code that was generated for this domain, executing some statement T dependent on x and y.

We clearly see the lexicographic order on the variables in the generated code and the top level loop distribution at $x = 4$. On closer inspection, we notice that the loop bounds are roots of multivariate polynomials describing the index space: We can rewrite the constraints of the domain into the form $p \geq 0$ with p being the rewritten polynomial. The following table shows the rewritten constraints and their roots i.e. the solutions of $p = 0$.

Constraint		Polynomial constraint		Roots
$1 \leq x$	$\Leftrightarrow$	$x - 1 \geq 0$	$\leadsto$	$x = 1$
$x \leq 7$	$\Leftrightarrow$	$-x + 7 \geq 0$	$\leadsto$	$x = 7$
$1 \leq y$	$\Leftrightarrow$	$y - 1 \geq 0$	$\leadsto$	$y = 1$
$y \leq 9$	$\Leftrightarrow$	$-y + 9 \geq 0$	$\leadsto$	$y = 9$
$(y-4)^2 - 3x + 12 \geq 0$	$\Leftrightarrow$	$(y-4)^2 - 3x + 12 \geq 0$	$\leadsto$	$y = 4 \pm \sqrt{3x - 12}$

[1]Example taken from [Grö09].

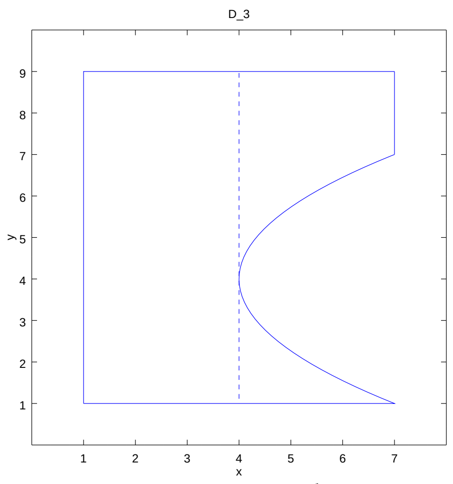


Figure 2.1.: Non-convex domain D_3

```
for (x = 1; x <= 4; ++x) {
  for (y = 1; y <= 9; ++y) {
    T(x,y);
  }
}
for (x = 5; x <= 7; ++x) {
  for (y = 1; y <= ⌊4 − √(3x − 12)⌋; ++y) {
    T(x,y);
  }
  for (y = ⌈4 + √(3x − 12)⌉; y <= 9; ++y) {
    T(x,y);
  }
}
```

Listing 2.3: Code generated for the domain D_3

As one can see, the roots directly describe the loop bound expressions. This insight hints that we should find algorithms for code generation that extract those loop bounds in a lexicographically correct order. As it turns out, this problem can be solved using Cylindrical Algebraic Decomposition, a technique we will explain in Section 2.3.

2.1.3. Occurence of non-linearities in practice

By now one might suggest that these non-linearities occur very rarely.

However, this is not true, since most often such non-linearities occur when a non-linear schedule is applied as the space-time mapping during the optimizing transformations in the Polyhedron model. Applying such non-linear schedules is especially common in the case of automated schedule exploration which aims to find the best schedule.

Sometimes, loops also contain non-constant strides depending on outer loop iterators, i.e. a loop header might be of the form `for (x = 0; x <= n; x += i)` where i is a surrounding loop iterator. One possible application scenario for this case might be general implementations of convolutions in image processing. During normalization of the loop header (what is essential for further optimizations) this loop would be transformed into one of the form `for (j = 0; j*i <= n; ++j)` where each x in the original loop body is replaced with the non-linear expression $j * i$.

A further source of non-linearities we want to focus on especially is the parametric tiling which we will describe in detail in the next section. In this case, the runtime parameters will generate non-linearities in the loop body to reconstruct the original loop variable value out of a combination of two loop iterators in the tiled system.

2.2. Tiling

Tiling is an often used optimization to coarsen the grain of parallelism, improve cache utilization, exploit NUMA architectures or minimize communication in distributed computations [Xueoo].

In this optimization, the iteration domain is partitioned into multiple *tiles* of a specific size, effectively doubling the number of variables. In case of cache utilization this may be beneficial for memory prefetching, or in case of distributed systems it narrows the amount of communication which is no longer performed from point to point, but rather from tile to tile.

Figure 2.2 shows an example of tiling applied onto a triangular index space using tiles of a rectangular shape. In this picture, each dot represents a single execution of the loop body. Note that this image does not take dependencies into account which will commonly result in a skewing of the index space.

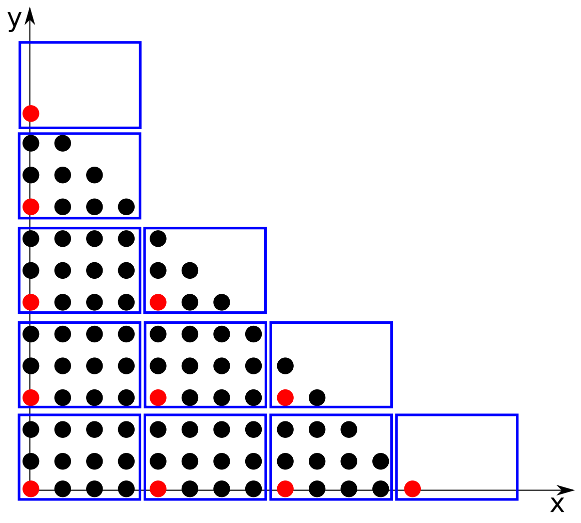


Figure 2.2.: Rectangular tiling of a triangular index space

In this case, each tile (depicted blue) has a width of 4 points and a height of 3 points. The red dots at the lower-left corner of each tile show the local origins of the tiles i.e. the first points that will be executed of the respective tile and these are also the points where communication happens.

Due to the lexicographic ordering, the iteration process in this example would start with the tile on the lower-left in the point $(0, 0)$. After this tile is completely processed, the tile directly above it will be executed and so on, until the *vertical* tiles are all processed. Then the execution continues at the second-left tile on the bottom.

Besides, a tile does not always need to be executed fully. The example also shows five tiles of which only a part of the tile points are executed. But note that there is no empty tile which would represent empty loop nests.

2.2.1. General definition

As [AI91] introduced, the tiling of an iteration domain can be expressed by stating a *lattice* matrix defining the translation from one tile to another, a matrix defining the shape of a single tile and equations that define how the original loop iterators shall be replaced with a combination of the new tile and loop iterators.

We follow the refined definition of this system given in [Grö09].

Definition 2.8 (Tiling). *Let M be a matrix and q be a vector describing the index set polyhedron P according to Definition 2.4 on page 3. Furthermore, let T be a matrix describing the shape of a single tile and let A be the lattice.*

Then for points $x = (x_1, \dots, x_n)^T \in P$ the tiled system can be expressed in terms of tile coordinates $t = (t_1, \dots, t_n)^T$ and point coordinates $o = (o_1, \dots, o_n)^T$ as follows:

$$Mx \geq q, \qquad To + t \geq 0, \qquad x = At + o$$

Remark: The tile coordinates t enumerate the tiles themselves and the point coordinates o enumerate the points inside a single tile.

As we see, tiling effectively doubles the number of variables of the original system of inequalities.

2.2.2. Parametric Tiling

A severe limitation in the common polyhedral model is that the lattice could not contain runtime parameters. More explicitly, all tile sizes have to be known at compile-time.

This has multiple disadvantages. One of the most prominent disadvantages is that tiling is often used in load balancing, and fixed tile sizes require that a user of a tiling library specifies these parameters explicitly and compiles the entire code again.

Another disadvantage is that non-parametric tilings cannot efficiently be evaluated using auto-tuning compilers for the same reason.

These problems can be solved when allowing parameters in the lattice, but this comes at the price of computational expensive machinery like a generalied Fourier-Motzkin method, or the usage of Cylindrical Algebraic Decomposition on which we focus.

2.2.3. Parallelepiped Tiling

A special subclass of tiles are those whose opposite sides are parallel. Common tile shapes that belong to this class are rectangular and parallelogram-shaped tiles in a two-dimensional tiling.

These tile shapes share the nice property that they can be expressed even easier.
Let $K = (v_1 | \dots | v_n) \in \mathbb{Q}^{n \times n}$ be a matrix whose column vectors are the spanning vectors of a tile. As such, all $v_i, i = 1, \dots, n$ are linearly independent and therefore we know that in this case K is always invertible.

As another nice property, we can easily describe that and how tile sizes depend on parameters $p = (p_1, \dots, p_n)^T$. Then the tiled system can be defined similarly to above by the new system of inequalities

$$Mx \geq q, \qquad 0 \leq K^{-1} o \leq \begin{bmatrix} p_1 - 1 \\ \vdots \\ p_n - 1 \end{bmatrix}, \qquad x = At + o \tag{2.1}$$

where the lattice is given through

$$A = K \operatorname{diag}(p_1, \dots, p_n) = K \begin{bmatrix} p_1 & & \\ & \ddots & \\ & & p_n \end{bmatrix}.$$

The middle part of the system (2.1) also forces that $1 \leq p_i, i = 1, \dots, n$ i.e. that all parameters are greater than zero. This makes all tiles non-empty.

As a remark, the above definition of the system of inequalities was inspired by [Grö09]. However, in the definition given there, the lattice matrix A was computed by the matrix-vector product $K \cdot (p_1, \dots, p_n)^T$ what is incorrect as A has to be a quadratic matrix rather than a vector. Furthermore, the middle part of the system (2.1) differed from our definition, as the original definition in [Grö09] produced a wrong result.

2.2.4. A parametric parallelepiped tiling example

Let us reconsider the example shown in Figure 2.2 on page 7.
The original triangular index set is described by the inequalities

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -1 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ -p \end{bmatrix}$$

for a parameter $p \in \mathbb{Z}$ which is set to 12 in the picture.
Just as depicted, we use tiles of a rectangular shape depending on parameters $w, h \in \mathbb{Z}$. In the picture we have $w = 4$ and $h = 3$.
The spanning vectors of a rectangle are $v_1 = (1, 0)^T$ and $v_2 = (0, 1)^T$.
This defines our matrices

$$K = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \qquad A = K \operatorname{diag}(w, h) = \begin{bmatrix} w & 0 \\ 0 & h \end{bmatrix}.$$

All together, our tiled system is given through

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -1 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ -p \end{bmatrix}, \qquad 0 \leq \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} o_1 \\ o_2 \end{bmatrix} \leq \begin{bmatrix} w-1 \\ h-1 \end{bmatrix}, \qquad \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} w & 0 \\ 0 & h \end{bmatrix} \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} + \begin{bmatrix} o_1 \\ o_2 \end{bmatrix},$$

or equivalently through

$$\begin{array}{lclcl} 0 & \leq & wt_1 + o_1 & & \\ 0 & \leq & ht_2 + o_2 & \leq & p - wt_1 - o_1 \\ 0 & \leq & o_1 & \leq & w - 1 \\ 0 & \leq & o_2 & \leq & h - 1 \end{array}$$

2.3. Cylindrical Algebraic Decomposition

In the previous sections, we described problems which do occur in practice but cannot be treated with polyhedral code generation.
Since these cases cannot be handled by polyhedral code generation, other techniques have to be developed. A solution to this problem was found by [Grö09] and this solution is based on *Cylindrical Algebraic Decomposition* (CAD), allowing arbitrary polynomial expressions to occur.
In this section, we will introduce the reader to the realm of this type of decomposition and illustrate how this can be used for code generation in the generalized polyhedron model.

2.3.1. Introduction

In the generalized polyhedron model, we have a description of index sets $\mathcal{I} \subset \mathbb{Z}^n$ based on a system of inequalities $\mathcal{P}$ with each having the form $p \sim 0$ where $p \in \mathbb{Q}[x_1, \ldots, x_n]$ can be arbitrary polynomials and $\sim$ is any of the binary relations $\{<, \leq, =, \neq, \geq, >\}$.
Our goal is to generate code which enumerates all points in the index set. This is equivalent to answering the question $\exists x \in \mathbb{R}^n : \forall p \in \mathcal{P} : p(x) \sim 0$ i.e. for what points $x \in \mathbb{R}^n$ all inequalities describing the index set are satisfied.

To answer this questions we would like to get descriptions of what values each variable $x_i \in \mathbb{R}$ can take. This can be achieved by *quantifier elimination* over the reals. More precisely, we want to create formulae involving our variables which do not contain any quantifiers anymore and are equivalent to the original constraints.

This method is important in many scientific applications and was originally introduced by Alfred Tarski in 1948. He showed that quantifier elimination over real closed fields is decidable, whereas he also stated that this procedure is not decidable for any theories that involve arithmetic over integers or rationals [Tar48].

Over the years, improvements to his original method were made, but they all were impractical even for simple examples. A major step in improving it was made by George Collins in 1975 with the invention of the method of *Cylindrical Algebraic Decomposition* [Col75].

In the next part, we will briefly introduce Collins' algorithm for computing a CAD as proposed in his work.

2.3.2. Computation of a CAD

Before we explain how a CAD is computed, we have to introduce some basic terminology used.

As we already stated in Section 2.1.2, the loop bounds we want to compute based on the index set description using polynomial constraints are the roots of these possibly multivariate polynomials. Therefore, we start by defining terms representing these roots.

Definition 2.9 (Algebraic Number). *Let $n \in \mathbb{N}$ and let $\mathbb{Q}[x_1, \ldots, x_n]$ denote a polynomial ring with a polynomial having rational coefficients in variables $x_1, \ldots, x_n$ and let $0 \neq p \in \mathbb{Q}[x_1, \ldots, x_n]$ be a polynomial.*

A number a is called an algebraic number *if it is a root of the polynomial p i.e. if $p(a) = 0$. If a is a real root of p, it is called a* real algebraic number.

The set $\mathbb{A} = \{a \mid p(a) = 0\}$ is called the set of algebraic numbers *of p.*

As a simple conclusion, we know that every rational number is a root of a rational polynomial, since every rational number $r = a/b$ is a root of the polynomial $b \cdot x - a$. But not every real number is algebraic: The real number $\sqrt{2}$ is algebraic because it is a real root of the polynomial $x^2 - 2$, but the real number π cannot be algebraic.

Since any number that cannot be a root of a polynomial with integral or rational coefficients is transcendental, we know that every transcendental number (e.g. π) is not algebraic.

Both statements together yield the set inclusion hierarchy $\mathbb{Q} \subsetneq \mathbb{A} \subsetneq \mathbb{R}$.

Since the loop nests we want to generate somewhat divide the field $\mathbb{R}^n$, we define notions for that too.

Definition 2.10 (Section/Sector). *Let $R \subset \mathbb{R}^n$ be a so-called* region *i.e. a non-empty connected set. Let $f, f_1, f_2\colon R \to \mathbb{R}$ denote continuous functions.*

Then the set

$$\{(x, f(x)) \mid x \in R\}$$

*is called a f-*section *over the cylinder $R \times \mathbb{R}$. Furthermore, the set*

$$\{(x, y) \mid (x, y) \in R \times \mathbb{R} \wedge f_1(x) < y < f_2(x)\}$$

*is called a (f_1, f_2)-*sector *over the given cylinder. Note that $f_1 = -\infty$ or $f_2 = +\infty$ are possible too.*

We can also interpret the definitions of sections and sectors graphically: Sections define a single connected "line" across the field $\mathbb{R}^n$, while the sectors denote the gap between two sections.

Definition 2.11 (Decomposition). *Let $R \subset \mathbb{R}^n$.*
A decomposition *of R is a collection of sets $R_1, \ldots, R_n, n \in \mathbb{N}$ for which the equations*

$$R = \bigcup_{i=1}^{n} R_i \quad \text{and} \quad R_i \cap R_{i+1} = \emptyset, \quad i = 1, \ldots, n-1$$

hold. So, mathematically, a decomposition is a partitioning.

This definition of a decomposition being a partitioning mathematically is pretty clear. However, since we aim to search for the boundaries of the indiviual partitioning subsets and we stated earlier that these boundaries are roots of polynomials, we refine our definition.

Definition 2.12 (Algebraic Decomposition). *A set is called* semi-algebraic *if it can be described by quantifier free polynomial (in-)equalities. A decomposition is called* algebraic *if each region of it is a semi-algebraic set.*

Since we want to construct a decomposition based on roots of polynomials, it is obvious that we partition the current dimension into disjoint parts. This gives rise to the following definition.

Definition 2.13 (Stack). *Let $R \subset \mathbb{R}^n$ be a region and let $D = f_1, \ldots, f_m, m \in \mathbb{N}$ be a set of continuous functions $f_i, i = 1, \ldots, m$ such that $f_1(x) < \ldots < f_m(x), x \in R$.*
We call D a stack *over R, which is defined by a decomposition of $R \times \mathbb{R}$ consisting of*

- *the sector $(-\infty, f_1)$, and*
- *all sectors (f_i, f_{i+1}) for $i = 1, \ldots, m-1$, and*
- *all sections f_i for $i = 1, \ldots, m$, and*
- *the sector $(f_n, +\infty)$.*

Next, we can state when a decomposition is cylindrical.

Definition 2.14 (Cylindrical Decomposition). *Let D be a decomposition of a set $R \subset \mathbb{R}^n$.*
D is called cylindrical *if for*

- *$n = 1$ we have that $D = -\infty < r_1 < r_2 < \ldots < r_{m-1} < r_m < +\infty, r_i \in \mathbb{R}$, and for*
- *$n > 1$ it holds that there exists a cylindrical decomposition D' of $\mathbb{R}^{n-1}$ such that D contains a stack for each region of D'.*

In other words, a decomposition is cylindrical if it is partitioned into stacks on every dimension.

Finally, we know all terms to define the decomposition we want to compute.

Definition 2.15 (Cylindrical Algebraic Decomposition). *A* Cylindrical Algebraic Decomposition *is a decomposition which is both algebraic and cylindrical.*

Based on the above defined terminology, a Cylindrical Algebraic Decomposition is a cylindrical and algebraic partitioning of the field $\mathbb{R}^n$ into a finite number of so-called *cells*. The algorithm we explain in this section creates a *sign-invariant* decomposition i.e. the sign of a polynomial over a cell is constant, which is the key property for our code generation to enumerate the statements in the correct lexicographic order. That is, because in this case the sign of a polynomial over a CAD cell is constant and hence the polynomial cannot cross another polynomial or itself, thus a decomposition is generated which contains at least all roots and intersections connected with the iteration domains.

Such a partitioning is computed following three phases:

1. The projection phase.
2. The base phase.
3. The lifting phase.

The projection phase

The first phase of a CAD computation is the projection phase. Its goal is to take the current set of polynomials $P^{(n)} \subset \mathbb{Q}[x_1, \ldots, x_n]$ in n variables and project it to a new set $P^{(n-1)} \subset \mathbb{Q}[x_1, \ldots, x_{n-1}]$ in $n-1$ variables, which have a constant number of real roots over certain regions i.e. which are delineable.

Definition 2.16. *Let $p \in \mathbb{Q}[x_1, \ldots, x_n]$ be a polynomial. p is called* delineable *[Col75] over a region R if the real variety i.e. the set $\{x \in \mathbb{Q}^n \mid p(x) = 0\}$ consists of only finitely many disjoint sections. A set of polynomials is called delineable if every polynomial in it either vanishes (i.e. is zero) or is delineable over R.*

The property of delineability is essential for the CAD algorithm as proposed by Collins to create cells that are either identical or disjoint, hence resulting in a cylindrical decomposition.

For computing a Cylindrical Algebraic Decomposition of $\mathbb{R}^n$ the projection phase is applied $n-1$ times.

There are multiple *projection operators* known, most prominently the operators of Collins, Hong, McCallum, and Brown. These operators require polynomial math machinery like principle subresultant coefficients, reducta sets, resultants or discriminants. Their precise definitions can be found in the literature [Col75, McC85, Hon90, Bro01].

Each operator in this series improves over the previous by reducing the number of projected polynomials. Let *projB* denote the set of projected polynomials by using Brown's projection operator, *projM* the projection set by McCallum, *projH* the projection set of Hong and *projC* the projection set by Collins. Essentially the set inclusion hierarchy $projB \subseteq projM \subseteq projH \subseteq projC$ holds [VKÁ17]. Since much work was invested in the past in optimizing this phase, current implementations are rather fast.

In this thesis, we will provide an example of the computation of a Cylindrical Algebraic Decomposition and explain our implementation of this method later. In both cases, we will assume that Brown's projection operator is used.

The base phase

After the projection phase, we want to compute a CAD of the field $\mathbb{R}^1$. This is rather easy because we only have to compute real roots of the univariate polynomials in $P^{(1)}$. Let $r_1, r_2, \ldots, r_l$ denote the $l \in \mathbb{N}$ real roots of these polynomials.

Then a Cylindrical Algebraic Decomposition of $\mathbb{R}^1$ is given by the stack consisting of

- the sector $(-\infty, r_1)$, and
- the sectors (r_i, r_{i+1}) for $i = 1, \ldots, l-1$, and
- the sections r_i with $i = 1, \ldots, l$, and
- the sector $(r_l, +\infty)$.

Like the projection phase, this part is quite fast as there are many well-known algorithms on how to efficiently compute the real roots of univariate polynomials.

In this phase, we also create $2l + 1$ test points $t_1, \ldots, t_{2l+1} \in \mathbb{A}$ such that

$$-\infty < t_1 < t_2 = r_1 < t_3 < t_4 = r_2 < t_5 < \ldots < t_{2l-1} < t_{2l} = r_l < t_{2l+1} < +\infty.$$

The lifting phase

After both the projection and the base phase we encounter the computationally most expensive phase, namely the lifting (or extension) phase. This part re-uses CADs computed on level $i-1$ and constructs a CAD on level i, $i = 2, \ldots, n$. This is repeated until a CAD of $\mathbb{R}^n$ is computed.
The following steps must be done for lifting a CAD from level $i-1$ to level i:

1. For each test point $t \in \mathbb{A}^{i-1}$ constructed on level $i-1$, insert this point into all i-variate polynomials at this projection level, which results in a set of univariate polynomials.
2. Accumulate all real roots of these univariate polynomials.
3. Create another set of test points $t' \in \mathbb{A}^i$ from these roots similar to the base phase.

The result of this phase is a set of n-dimensional test points and descriptions of their respective cells. These test points then are used to determine whether its surrounding cell is part of the decomposition or not, by checking if the original constraints are fulfilled after inserting the test point.
As one can imagine, the lifting phase represents a tree which gets very large very fast. Our specific goal is to optimize this phase as this is the most expensive part of the computation.

2.3.3. Example

Now we want to apply the algorithm described above to a concrete example. For this, let us reconsider the example shown in Figure 2.1 on page 6.
So, the input to the algorithm is the system of polynomial constraints given through

$$1 \leq x \leq 7 \quad \wedge \quad 1 \leq y \leq 9 \quad \wedge \quad (y-4)^2 - 3x + 12 \geq 0.$$

For this example, we choose the variable ordering $y \rightarrow x$. Furthermore, we decide to use Brown's projection operator since it produces the least number of projected polynomials of all considered projection operators.
After applying the operator, we get the following projections:

Level 2 (y):	$x-1, x-7, y-1, y-9, (y-4)^2 - 3x + 12$
Level 1 (x):	$x-1, x-7, x-4, 3x-37$

The next step is to apply root isolation to the polynomials on level 1 (the base phase). This yields the real roots $1, 4, 7, \frac{37}{3}$.
As described earlier, we now construct test points based on these roots. These points are:

$$t_1 = 0, t_2 = 1, t_3 = 2, t_4 = 4, t_5 = 5, t_6 = 7, t_7 = 8, t_8 = \frac{37}{3}, t_9 = 13$$

Now we go into the lifting phase. In this phase, we have the set of polynomials in y with their respective multivariate roots. In our example we have the multivariate roots (for y):

$$y = 1, \quad y = 9, \quad \text{and} \quad y = 4 \pm \sqrt{3x - 12}.$$

The further things of this phase are visualized via the following table. In this phase, we have to take each test point of the base phase (first column) and substitute it into the polynomials on level 2, resulting in a new set [2] of univariate polynomials in y (second column). Then, we have to isolate the real roots of all these new polynomials again and create new test points. For brevity, we just state the set of real roots.

Test point	**New polynomials**	**Real roots**
$t_1 = 0$	$\{y-1, y-9, (y-4)^2+12\}$	$\{1, 9\}$
$t_2 = 1$	$\{y-1, y-9, (y-4)^2+9\}$	$\{1, 9\}$
$t_3 = 2$	$\{y-1, y-9, (y-4)^2+6\}$	$\{1, 9\}$
$t_4 = 4$	$\{y-1, y-9, (y-4)^2\}$	$\{1, 4, 9\}$
$t_5 = 5$	$\{y-1, y-9, (y-4)^2-3\}$	$\{1, 4 \mp \sqrt{3}, 9\}$
$t_6 = 7$	$\{y-1, y-9, (y-4)^2-9\}$	$\{1, 7, 9\}$
$t_7 = 8$	$\{y-1, y-9, (y-4)^2-12\}$	$\{1, 4 \mp \sqrt{12}, 9\}$
$t_8 = \frac{37}{3}$	$\{y-1, y-9, (y-4)^2-25\}$	$\{-1, 1, 9\}$
$t_9 = 13$	$\{y-1, y-9, (y-4)^2-27\}$	$\{4-\sqrt{27}, 1, 9, 4+\sqrt{27}\}$

Now we have a section/sector based decomposition of the space $\mathbb{R}^2$ and according test points. Next, we have to decide what CAD cells actually are part of our index space. We do so by checking if the original constraints are fulfilled for the respective test point.

To keep things short, we only show this for the point $x = t_2 = 1$.

According to the real roots for that point, we construct test points

$$t_{2,1} = 0, t_{2,2} = 1, t_{2,3} = 4, t_{2,4} = 9, t_{2,5} = 10$$

for the y dimension.

We determine which test points $t = (x, y) = (1, t_{2,i}), i = 1, \ldots, 5$ fulfill our original constraints. It turns out that the points $(1, 1), (1, 4)$ and $(1, 9)$ do so. Since $(1, 1)$ and $(1, 9)$ are sections and $(1, 4)$ is a test point for a sector between those two, we can infer that our original constraints are fulfilled for points (x, y) which fulfill $(x = 1 \wedge 1 \le y \le 9)$.

By applying this procedure to every constructed test point, we obtain the final, quantifier free formula

$$\begin{aligned} &(1 \le x \le 4 \wedge 1 \le y \le 9) \\ \vee &\left(4 < x < 7 \wedge \left(1 \le y \le 4 - \sqrt{3x-12} \vee 4 + \sqrt{3x-12} \le y \le 9\right)\right) \\ \vee &(x = 7 \wedge (y = 1 \vee 7 \le y \le 9)) \,. \end{aligned}$$

One can easily prove that this formula describes the same set of of points as the solution of the original quantified formula

$$\exists x\colon \exists y\colon \left(1 \le x \le 7 \wedge 1 \le y \le 9 \wedge (y-4)^2 - 3x + 12 \ge 0\right).$$

[2]Note that constant polynomials can be discarded from this set as they do not have roots. The only one that does so is the constant 0 polynomial, which has an infinite amount of roots, but this is of no use for computing a CAD, therefore we discard it too.

2.3.4. Algorithmic Complexity

In the first method of computing a CAD published by Collins in [Col75], he stated that his method for quantifier elimination on a formula φ given in prenex form [3] has an algorithmic complexity (i.e. an upper bound for the computation time) of

$$(2n)^{2^{2r+8}} m^{2^{r+6}} d^3 a,$$

where r is the number of variables, m is the number of polynomials, n is the maximum degree of any polynomial in any variable, d is the maximum length of any integer coefficient and a is the number of atomic formulas occuring in φ.

In terms of the big-O-notation, we see that his method lies in $\mathcal{O}\left(2^{2^r}\right)$. We say that the computation of the CAD is *doubly exponential* in the number of variables r.

Although several optimizations were applied over the past four decades, only some exponents of this bound could be lowered and the doubly exponential nature of this method still applies.

A common argument in this discussion is that one could also use Fourier-Motzkin elimination. However, even the simple elimination procedure is already doubly exponential in the number of variables [Sch86]. Additionally, to treat non-linearities one needs a generalized Fourier-Motzkin elimination method as described in [Grö09], which includes case-distinctions in general and hence makes the computational effort even worse.

[3] A prenex formula is of the form $Q_1 x_1 : \dots Q_n x_n : F$ where Q_i are quantifiers, x_i are variables bound by the quantifiers, and F is a formula which does not contain any quantifiers.

Chapter 3

Implementation and Optimizations

In the previous chapter we discussed the motivation why we want to generalize code generation to arbitrary polynomial constraints, and we introduced the well-known method of a Cylindrical Algebraic Decomposition.

In this chapter, we want to apply this method to real code generation. In the first section, we will state a concrete code generation algorithm which is able to generate code for index sets described by arbitrary polynomials based on Cylindrical Algebraic Decomposition.

The following sections will deal with general algorithmic enhancements to improve the performance of code generation followed by optimizations tuned towards special cases which occur often in practice.

3.1. Basic Code Generation

While in the previous sections we introduced the concept and computation procedure for a Cylindrical Algebraic Decomposition, we now apply this to our code generation task. We do so by embedding this computation into an algorithm capable of generating both case-distinctions on parameters and loops for arbitrary polynomial bounds based on a CAD computation.

As already mentioned, we use the freely available library SMT-RAT. This library consists of multiple tools we can use such as the projection operators and projection phase implementations and utilities for real root isolation of univariate and multivariate polynomials as well as utilities for arithmetic on real algebraic numbers.

It also contains an implementation of a partial lifting phase for the Cylindrical Algebraic Decomposition. However, this implementation was specialized for a *partial CAD*, i.e. the lifting stops immediately after one satisfied sample point is found. This is not useful for our purposes. In a first attempt, we did implement a lifting strategy in the library which does the full lifting. Unfortunately, we could not figure out how we can convert the representations of a CAD cell in the library into our structure of loops or parameters. Because of this reasons, we decided to implement the lifting phase by ourselves.

The code generation procedure which represents the lifting phase in pseudocode is shown in Algorithm 1 on the next page as derived by [Grö09].

It should be noted that this algorithm reflects the base and lifting phases of the CAD computation. An according projection phase has to be done prior to calling this function.

As an overview, the following parameters/symbols appear in the code:

Symbol	Interpretation
S	The current region to construct a stack over.
domains	List of index sets.
t	Current test point.
n	Number of variables i.e. loop iterators.
q	Number of parameters.
dim	Current recursion level, starts with 1 and increases up to $n + q + 1$.

Algorithm 1 Code generation procedure based on CAD

```
function CODE_GEN(S, domains, t, dim, $n$, $q$)
    if dim = $n + q + 1$ then
        code ← ""
        for all $d \in$ domains do
            if t $\in d$ then
                code ← code + "$T_d$;"
            end if
        end for
        return code
    end if

    Let $f_1, \ldots, f_r$ be the sections defining a stack over $S$
    $f_0 \leftarrow -\infty$
    for $i = 1, \ldots, r$ do
        code ← code + SECTOR_CODE($f_{i-1}$, $f_i$,domains, t, dim, $n$, $q$)
        code ← code + SECTION_CODE($f_i$, domains, t, dim, $n$, $q$)
    end for
    code ← code + SECTOR_CODE($f_r$, $+\infty$, t, dim, $n$, $q$)
    return code
end function

function SECTOR_CODE($f_1$, $f_2$, domains, $t$, dim, $n$, $q$)
    $t' \leftarrow (t$, rational_between($f_1(t), f_2(t)$))
    inner ← CODE_GEN(sector(S, $f_1$, $f_2$), domains, $t'$, dim +1, $n$, $q$)
    if inner = "" then
        return ""
    end if
    if $dim \leq q$ then
        head ← "if ($p_{dim} > f_1$ and $p_{dim} < f_2$)"
    else
        if $f_1 \neq -\infty$ then $f_1 \leftarrow \lfloor f_1 \rfloor + 1$ end if
        if $f_2 \neq +\infty$ then $f_2 \leftarrow \lceil f_2 \rceil - 1$ end if
        head ← "for ($x_{dim-q} = f_1; x_{dim-q} \leq f_2; x_{dim-q}$++)"
    end if
    return head + "{" + inner + "}"
end function

function SECTION_CODE($f$, domains, $t$, dim, $n$, $q$)
    inner ← CODE_GEN(section(S, root), domains,($t$, $f(t)$), dim+1, $n$, $q$)
    if inner = "" then
        return ""
    end if
    if $dim \leq q$ then
        head ← "if ($p_{dim}$ == $f$)"
    else
        head ← "for ($x_{dim-q} = \lceil f \rceil; x_{dim-q} \leq \lfloor f \rfloor; x_{dim-q}$++)"
    end if
    return head + "{" + inner + "}"
end function
```

Additionally, we use the utility functions `section`, `sector` and `rational_between`. The first two of them define a section or sector respectively out of the current region and the boundaries, while `rational_between` computes any rational point between its two parameters.

Now we describe the indiviual functions appearing in Algorithm 1 on the facing page.

We start with the function CODE_GEN.

The first conditional statements check if we are in the base case of the recursion, i.e. if we already have a CAD of the entire space given. It then decides based on the constraints of the iteration domains if statements should be emitted for that respective domain.

If we are not in the base case yet, we compute the sections defining a stack over the current region S. Then we start the recursive calls by (possibly) creating code for each sector and section defining that stack, and add the result to our code.

Finally, the result is returned.

The function SECTION_CODE generates code for a single section. To do so, it invokes the code generation procedure recursively with a new test point over the current region including the current section point. If code was generated in this call, we know that the inner loops or parameters do not carry empty executions. Then, based on the current recursion level, we create either code for a parameter or a loop section.

Similar to this function we have the function SECTOR_CODE, which generates code for a single sector. It basically does the same as SECTION_CODE, with the difference that we must provide a new test point somewhere between the two limits given, and that the loops and parameters now carry more complex comparisons.

As short remarks, we will state that the algorithm actually behaves correctly.

Proposition 3.1 (Correctness)**.** *The code generation algorithm as stated in Algorithm 1 is correct in the sense that it generates code which executes exactly the integral points of its input sets in the correct lexicographic order.*

Proof. It can be seen easily that this represents the base and lifting phases of the computation of a CAD and these are correct by algorithmic proofs given in [Col75]. The lexicographic order is given by the way we construct our stack and is therefore correct. Finally, the correct usage of the $\lceil\cdot\rceil$ and $\lfloor\cdot\rfloor$ functions in the code ensure that all necessary integral points are executed. □

Proposition 3.2 (Termination)**.** *The code generation algorithm in 1 on the preceding page always terminates.*

Proof. This is a direct consequence of the fact that a CAD computation has an upper bound for its computation time [Col75]. □

Remark 1: The algorithm proposed is able to generate code for unlimited sectors. While this should not happen for loops, it can and will happen quite often for parameters. In this case we have a case of a parameter being bigger than some value but not limited to an upper bound.

Remark 2: In our real implementation in the programming language C++ we do not use plain strings as indicated in the pseudocode, but rather we manipulate our own abstract syntax tree which carries all necessary information. As a consequence, we do not return empty strings if no code shall be generated, but propagate this information using boolean flags.

3.2. A code generation example

We can now apply this code generation algorithm to our introductory problem described in Section 2.3.3 and show that the result is (after simplification) exactly the code depicted in Listing 2.3 on page 6.

We start our description from the base phase of the CAD computation for this example, as already explained in Section 2.3.3. In that section we already computed test points

$$t_1 = 0, t_2 = 1, t_3 = 2.5, t_4 = 4, t_5 = 5.5, t_6 = 7, t_7 = 8, t_8 = \frac{37}{3}, t_9 = 13$$

for the $\mathbb{R}^1$ decomposition. These test points are depicted as red diamonds on the x-axis in Figure 3.1. Please note that Figure 3.1 only shows relevant test points.

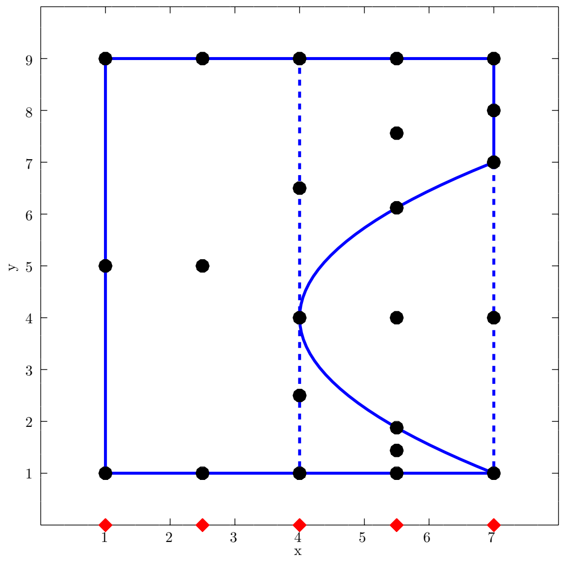


Figure 3.1.: Index set D_3 including most test points

In the lifting step we have to construct stacks over each of these test points. As described earlier, we have to isolate the roots of the newly computed univariate polynomials and create test points. These test points in the two-dimensional plane are depicted as black circles. Note how these points are aligned over their respective one-dimensional test point. This property gives a visual interpretation of the phrase of constructing a stack *over* a test point.

Since we have already reached the last level in this example, we now have to decide what two-dimensional test points fulfill all constraints defining our original domain.

The following table summarises this paragraph. Note we again do not show test points which are clearly outside of our iteration domain. In the implemented code generation however there are some more test points, for which in the base case of the recursion it can be decided that they do not belong to the index set.

Test point t	**Test points** t' ***over*** t	t' **fulfills all original constraints?**
$t_2 = 1$	$(1, 1)$	✓
	$(1, 5)$	✓
	$(1, 9)$	✓
$t_3 = 2.5$	$(2.5, 1)$	✓
	$(2.5, 5)$	✓
	$(2.5, 9)$	✓
$t_4 = 4$	$(4, 1)$	✓
	$(4, 2.5)$	✓
	$(4, 4)$	✓
	$(4, 6.5)$	✓
	$(4, 9)$	✓
$t_5 = 5.5$	$(5.5, 1)$	✓
	$(5.5, \frac{5-\sqrt{4.5}}{2})$	✓
	$(5.5, 4-\sqrt{4.5})$	✓
	$(5.5, 4)$	↯
	$(5.5, 4+\sqrt{4.5})$	✓
	$(5.5, \frac{13+\sqrt{4.5}}{2})$	✓
	$(5.5, 9)$	✓
$t_6 = 7$	$(7, 1)$	✓
	$(7, 4)$	↯
	$(7, 7)$	✓
	$(7, 8)$	✓
	$(7, 9)$	✓

And for each test point which fulfills the original constraints of the domain, we have to create either a sector or a section code fragment as described in Algorithm 1 on page 18, depending on the fact if the test point defines a root of a used polynomial or not.

The fully generated code executing some statement T for our example is shown in Listing 3.1.

```
for (x = 1; x <= 1; ++x) {
  for (y = 1; y <= 1; ++y)  T;
  for (y = 2; y <= 8; ++y)  T;
  for (y = 9; y <= 9; ++y)  T;
}
for (x = 2; x <= 3; ++x) {
  for (y = 1; y <= 1; ++y)  T;
  for (y = 2; y <= 8; ++y)  T;
  for (y = 9; y <= 9; ++y)  T;
}
for (x = 4; x <= 4; ++x) {
  for (y = 1; y <= 1; ++y)  T;
  for (y = 2; y <= 3; ++y)  T;
  for (y = 4; y <= 4; ++y)  T;
  for (y = 5; y <= 8; ++y)  T;
  for (y = 9; y <= 9; ++y)  T;
}
for (x = 5; x <= 6; ++x) {
  for (y = 1; y <= 1; ++y)  T;
  for (y = 2; y <= ⌈4 − √(3x − 12)⌉ − 1; ++y) T;
  for (y = ⌈4 − √(3x − 12)⌉; y <= ⌊4 − √(3x − 12)⌋; ++y) T;
  for (y = ⌈4 + √(3x − 12)⌉; y <= ⌊4 + √(3x − 12)⌋; ++y) T;
  for (y = ⌊4 + √(3x − 12)⌋ + 1; y <= 8; ++y) T;
  for (y = 9; y <= 9; ++y) T;
}
```

```
for (x = 7; x <= 7; ++x) {
  for (y = 1; y <= 1; ++y) T;
  for (y = 7; y <= 8; ++y) T;
  for (y = 9; y <= 9; ++y) T;
}
```

Listing 3.1: Code generated for scanning domain D_3

As one can see, the generated code gets quite long. On closer inspection, we notice that many loops can be fused to shorten the generated program. The final result of the code generation for domain D_3 is shown in Listing 3.2.

```
for (x = 1; x <= 4; ++x) {
  for (y = 1; y <= 9; ++y)
    T;
}
for (x = 5; x <= 7; ++x) {
  for (y = 1; y <= ⌊4 − √(3x − 12)⌋; ++y)
    T;
  for (y = ⌈4 + √(3x − 12)⌉; y <= 9; ++y)
    T;
}
```

Listing 3.2: Generated code with fused loops scanning domain D_3

3.3. Influence of variable orderings and projection operators

Before we continue to state our different optimizations, let us briefly discuss the influence of the elimination order on variables as well as the influence of the used projection operator.

3.3.1. Influence of the variable ordering

Like when using the Fourier-Motzkin elimination method, one has to specify an ordering on the variables occuring in the system of inequalities. This ordering specifies what variables will be eliminated in subsequent steps.

Depending on the individual form of the constraints and how different variables are connected with each other, the elimination of a variable in a "bad" ordering can create lots of new polynomials, which makes things worse, especially considering the doubly exponential nature of the Cylindrical Algebraic Decomposition. In our experiments, we tried one of our test cases with several different variable orderings and all of our optimizations included, and the worst ordering resulted in elimination times being $20,000$ times slower than the best ordering.

During our experiments we observed that for the special case of parametric tiling, it is best to eliminate the loop iterator variables of the tiled loops in increasing order.

Definition 3.1 (Custom variable ordering)**.** *Let $n \in \mathbb{N}$ be the depth of a loop nest. Let t_i denote the loop iterator variables enumerating the tiles and let o_i denote the loop iterator variables enumerating the points inside the tile loops, $i = 1, \ldots, n$.*

Then we choose the variable elimination ordering $o_1 \prec t_1 \prec \ldots \prec o_n \prec t_n$, where the ordering $\prec$ defines the succeeding variable elimination.

This variable ordering is the fastest elimination order which respects the property that all point loops remain in their respective tile loops, although this ordering might not be of big practical use.

Another approach, suitable for the general case is the heuristic given by Brown [Bro04], in which the following steps (starting with the first step and breaking ties with the next) are applied:

For x, y being variables, let $x \prec y$ if

1. $\deg(x) \prec \deg(y)$, breaking ties with
2. $\text{tdeg}(x) \prec \text{tdeg}(y)$, breaking ties with
3. $\#\{t|t \text{ contains } x\} \prec \#\{t|t \text{ contains } y\}$.

In this heuristic, deg represents the overall degree of the variable, tdeg represents the highest total degree of terms in which the passed variable occurs, and $\#S$ represents the cardinality of the set S. One can easily verify that the third step of this heuristic matches with our heuristic in most cases.

There is ongoing research on this topic, with other strategies mentioned in [HEW+15]. However, these strategies require much more computational effort.

In our application, the basic elimination order has to be specified by the user. In case we use parametric parallelepiped tiling, then we use our elimination strategy given in Definition 3.1 on the facing page. For all other cases, the order provided by the user can be used, or the heuristic given by Brown may be used, which works suitably well. It should be noted that Brown's heuristic can achieve better performance even when used on tiled systems, however this depends on the tile shape and it violates the contract that each point loop is strictly inside its corresponding tile loop.

3.3.2. Influence of the projection operator

In Section 2.3.2 we already pointed out that there are multiple different projection operators, with each one improving its predecessor.

We also stated that this improvement is that the set of projected polynomials is smaller than the one of the preceeding operator.

But why is this an improvement?

The answer to this comes clear when re-considering the lifting phase of the Cylindrical Algebraic Decomposition. In this phase, we accumulate all real roots of the projected polynomials at a certain recursion level, and for each root we perform another chain of recursive calls which do essentially the same.

Simply said, the larger the set of projected polynomials, the more (cascading) recursive calls we have to make, which decrease performance.

That is why the choice of the projection operator is essential in improving the speed of code generation. Besides the fact that McCallums projection operator must add additional polynomials if the projected polynomials vanishes identically over zero [McC85], we choose the projection operator defined by Brown. This operator is not bound to this limitation and it generates much fewer polynomials in practice.

For our test case which consists of a rectangular index space which is tiled using parallelogram shaped tiles, we compared the number of projected polynomials between the operators of McCallum and Brown. The results are shown in Table 3.1 on the next page.

We can clearly see that McCallum produces a much bigger set of projected polynomials, at worst it produces about 20 times more polynomials. Besides the duration of the computation of the projection taking a much longer time, the lifting phase gets even worse considering the exponential nature of the lifting phase.

Projection	Level 6	Level 5	Level 4	Level 3	Level 2	Level 1
Brown	8	4	7	10	28	102
McCallum	8	6	14	31	160	1, 982

Table 3.1.: Number of projected polynomials using projections of Brown and McCallum

3.4. Reducing memory consumption

Our first optimization was to reduce the memory consumption of the code generator. This enables the usage of the generator for real world problems and increases performance, as less memory has to be transferred.

This optimization basically can be divided into two parts, the loop fusion and the memory allocation part.

3.4.1. Loop Fusion

As mentioned earlier, in practice many loops can be fused together. But this does not only reduce the size of the generated code, it further reduces memory consumption.

In our custom abstract syntax tree, we represent each such parameter conditional statement or loop using the following structure:

```
enum class LOP_FLAG : unsigned char {
    /* 8 bit masks */
};

using LOP_FLAG_INT = typename std::underlying_type<LOP_FLAG>::type;

class RootExpr {
    typedef unsigned char RootIndex;

public:
    UPoly polynomial;
    RootIndex index;
    carl::RealAlgebraicNumber<smtrat::Rational> ran;

    /* ... */
};

class LoP {
public:
    RootExpr lower, upper;
    LOP_FLAG_INT flags;
    unsigned int indentation;
    std::list<SmartPtr<LoP>> lop;
    std::list<std::string> stmts;

    /* ... */
};
```

Listing 3.3: AST used in our implementation

By just considering the size of these structures using the `sizeof` operator, on most machines each such `LoP` has a size of 296 bytes, which increases depending on how many statements (as strings) or inner `LoP`s it has.

Therefore, by fusing loops, we only need to adapt the lower bounds (or upper bounds, respectively) and some flags. Thus, the more loops we can fuse, the less memory is needed in our AST to represent the generated program.

What remains is the criterion of when loops or parameter conditionals can be fused.

Lemma 3.1. *Let l_1, l_2 denote two instances of* `LoP` *and w.l.o.g. assume that $l_1 \prec_{\text{lex}} l_2$, where $\prec_{\text{lex}}$ is the lexicographic ordering of the generated loops/conditionals, i.e. the order in which the current stack is generated.*

Then these two can be fused into one instance if the upper limit of l_1 is equal to the lower limit of l_2, their bodies (i.e. their inner `LoP`*s or statements) are equal, and if either l_2 represents a sector and l_1 represents a section, or vice-versa.*

Proof. For such instances, if l_2 is a sector (or a section, respectively) and l_1 is a section (or a sector, respectively), and if their limits match as described, we know that there cannot be any statements to execute between these two codes. If their bodies match too, then these fragments can be combined. □

3.4.2. Memory allocation

As we just saw, fusing loops can have quite an impact on the memory consumption and performance of our program. Another further consideration is the actual size of our objects.

We stated that each `LoP` consists of 296 bytes at minimum. Considering `C++` memory allocation, such objects are still considered small. It is a well-known problem that the standard memory allocator in `C++` is a thin wrapper around the pair of dated functions `malloc`/`free` in `C`, and that the latter pair of functions were initially designed to allocate or free large parts of memory at once.

Therefore, they are not best-suited for small object allocation. To cope with this, we decided to use Google's `tcmalloc` [1] memory allocator.

As benchmarks show, this memory allocator is well-suited for allocating small objects and not the worst allocator for bigger objects [2]. As we will see in Chapter 4, our implementation benefits from this choice too.

3.5. Pruning the lifting tree

As one can derive from our code generation procedure given in Algorithm 1 on page 18, the lifting phase of this CAD computation represents a tree where each node represents a function call. Since function calls are expensive in general, we try to avoid them as much as possible.

Therefore, if we can decide that a sample point cannot lie in a cell that belongs to our executed code, we can immediately stop the code generation for this subtree.

Of course, we must do so only if we can really ensure that this is fulfilled. This can be done if either every necessary information is known for a constraint to decide it, or if the partial information inserted into a constraint contradicts another constraint.

[1] `https://github.com/gperftools/gperftools`

[2] See `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`

We further distinguish between two types of such *early return* checks: If the sample point cannot belong to any domain, i.e. if it violates at least one (conjunctive) constraint of any domain, we can safely abort further code generation. If we have additional constraints which do not belong to the original domain constraints (e.g. if we have tiled our system), we can again stop our procedure for the current subtree if any of these constraints is violated for sure.

Furthermore, these checks can be done at multiple points during our code generation, pruning the lifting tree as soon and as often as possible.

3.6. Avoiding unnecessary exact arithmetic

Throughout the code generation we have to use real algebraic numbers in general. Although they are necessary for the computation of the roots, we do not need them for deciding whether a constraint is fulfilled or not.

Here, we can distinguish between checking against 0 for (in-) equality and checking against 0 using one of the relational operators $<, \leq, \geq, >$.

In the latter case, we can avoid real algebraic number arithmetic since we only need to determine the sign of the polynomial p evaluated at a test point t. This implies that in the case that we want to check $p(t) \sim 0$ with $\sim \in \{<, \leq, \geq, >\}$, it suffices to use a rational approximation of t and since arithmetic involving rational numbers is more efficient in general than dealing with real algebraic numbers, we can speed up our code generation further. For computing these rational approximations, we rely on functionality implemented in the library CArL which utilizes the GNU Multiple Precision Library to compute a very close approximation of the processed real algebraic number.

In the remaining case of $p(t) = 0$ or $p(t) \neq 0$, we explicitly need real algebraic arithmetic. As an example, if we want to check if $p(x) = x^2 - 2$ is exactly 0 at a test point $\sqrt{2}$, real algebraic number arithmetic indeed yields that this is the case, while a rational approximation would most likely return a result indicating the the constraint is not fulfilled, what is wrong.

In order to decrease the computation time further, many authors like Strzeboński describe variants of the Cylindrical Algebraic Decomposition which do not use real algebraic numbers for root computations, but rather root isolation using rational numbers and interval arithmetic. However, this only an approximation which has its own caveats. Because we want to ensure correctness in any case, we do not follow their approach. Their work is explained in detail in [Str06].

3.7. Parallelization

We already examined that the lifting phase resembles the construction of a tree whose nodes are either parameter cases or loops, and whose leaves are statements.

Because of its tree-like nature, it might be beneficial to parallelize the computation of the generated code. The idea is that each thread (limited to hardware ressources available) executes a branch of the computation. This parallelism can be nested too, if enough ressources are available.

Conceptually, only the back-writing of generated inner LoPs or statements must be protected by mutual exclusion, and all other parts can be done in parallel. However, due to excessive use of pooling datastructures in the library SMT-RAT and the fact that these datastructures are not protected against parallel access although SMT-RAT provides a thread-safe build, we have to ensure that these pools are accessed by one thread at a time, hence making all accesses to these pools mutually exclusive. Since the affected parts have to be called very often, we essentially reach the same performance as sequential. Our experiments in Chapter 4 confirm this.

During the implementation of our code generation, we tried two different parallel processing frameworks: OpenMP and the C++11 async method.

3.7.1. Parallelization using OpenMP

The first implementation used OpenMP [3] and its common parallelizing pragmas (like `#pragma omp parallel for`) among others. However, this implementation performed less well than our sequentiel implementation, mainly due to two reasons.

The first reason was that the parallelization of the code generation in its subtrees takes place in a loop which iterates over the accumulated roots of the currently projected polynomials. Since the number of these roots is not known at compile-time, OpenMP is unable to generate the most efficient code.

Secondly, OpenMP parallelized loops are known to not perform well when using nested parallelism [TTSY00], which is the case as the loop we want to parallelize calls its surrounding function recursively.

To overcome the poor performance of nested parallelism in `parallel for` loops, we switched to the OpenMP *tasking* model [4], which was explicitly designed to handle nested parallelism in recursive algorithms. Using this, we explicitly marked statements in our implementation which should execute in parallel and let OpenMP do the thread scheduling.

This method turned out to perform better than our first draft, but due to the mentioned synchronization points we essentially reached the same speed as when executing our program sequentially.

3.7.2. Parallelization using C++ Futures

The second framework we used for parallelization instead of OpenMP is the async mechanism native to C++11, giving thread-based parallelism using futures.

We derived a special type of lock which uses atomic flags for keeping track of its state and which yields CPU ressources to other threads if it is already locked. This turned out to be the most efficient type of lock for our application.

Listing 3.4 shows the most important parts of this implementation.

```
using LockT = Lock<LockingType::SPINLOCK>;
const int MAX_THREADS = std::thread::hardware_concurrency();
std::atomic<int> threadsActive = ATOMIC_VAR_INIT(0);

template<typename F, typename... Ts>
auto Async(F&& f, Ts&&... params) {
    using Policy = std::launch;
    auto launchPolicy = (threadsActive++ < MAX_THREADS ? Policy::async : Policy::deferred);
    return std::async(launchPolicy, std::forward<F>(f), std::forward<Ts>(params)...);
}
```

Listing 3.4: Excerpt of the parallelization using C++ futures

As one can see, based on hardware ressources available we either start a new thread or not. In the lifting functionality, a thread can be started if the user wishes to execute the program in parallel. After that functionality, we have to wait for the results of these futures, followed by decreasing our counter of active threads.

[3] http://www.openmp.org/
[4] http://www.openmp.org/specifications/

The intention of this method is to start new threads in early phases of the code generation which resemble high levels of the execution tree. This prevents the overhead of thread creation at lower levels and it simplifies the execution of lower levels. The disadvantage of this method is that we can only start a new thread when another thread terminates i.e. that the subtree executed by this thread was fully processed.

However, as stated neither method increased performance because of the limitations in the library SMT-RAT. Based on our experiments, we expect the best performance when using the async mechanism and we are looking forward that future work will reduce the amount of pooling data-structures in SMT-RAT and thus reduce the sychronization points, what is likely to result in better parallelization and actual runtime improvements.

3.8. Overapproximation for Tiled Systems

We already presented simple means for pruning the lifting tree by constraint contradition checking in Section 3.5. In that section, we also mentioned that we might have additional constraints which must be fulfilled.

One possibility is of course that the user explicitly specifies such constraints.

However, for the special case of parametric parallelepiped tiling we can actually calculate additional constraints. We already gave a definition for parametric parallelepiped tiling in Section 2.2.3, depending on a matrix $K \in \mathbb{Q}^{n \times n}$ and a parameter vector $p = (p_1, \ldots, p_n)^T$.

Based on that definition, we can immediately infer that $1 \leq p_i, i = 1, \ldots, n$ holds for every parameter. Taking this constraints into account, the lifting tree can be pruned instantly if a sample point component representing a parameter is non-positive.

What remains is the case of the tiled loop iterator combinations $t_i, o_i, i = 1, \ldots, n$. For this case, we can compute an overapproximation of the geometry of a single tile, i.e. a bounding box around it, which we want to describe in the following section.

3.8.1. Extreme Value Algorithm

The mentioned bounding box around a single tile does only depend on parameters and no longer on other loop iterators. Now by substituting these *extreme values* for the iterators in all constraints involving t_i, we get a new set of constraints where the constraints on variables t_i can be decided without the need of samples of the point loop iterators o_i.

This *Extreme Value Algorithm* is shown in Algorithm 2 on the facing page.

By replacing every point loop variable with their respective maximum intervals they can reach, we get a set of constraints on the tile loop iterators $t_i, i = 1, \ldots, n$ which depend only on possibly other tile loop iterators and on parameters. And by first computing a Cylindrical Algebraic Decomposition on the lower-dimensional system, we achieve maximum separation of tile loop variables too, making the resulting constraints easier to decide with partial information.

3.8.2. First overapproximation

We still need to provide the overapproximation of a single tile. We tried two approaches to solve this. As a first overapproximation, we designed Algorithm 3 on the next page.

Of course, we have to show that this algorithm does no harm.

Algorithm 2 Extreme value algorithm

Input:

- n - Number of index sets
- q - Number of parameters
- K - Tile shape matrix
- p - Parameter vector

```
procedure EXTREMEVALUEALGORITHM(constraints, $n$, $q$, $K$, $p$)
    Let $o$ be the vector of point loop variables.
    $p' \leftarrow (p_1 - 1, \dots, p_n - 1)^T$
    $(o_{i,\min}, o_{i,\max})_{i=1,\dots,n} \leftarrow$ OVERAPPROXTILE($o, K, p'$)
    cad ← CODE_GEN(∅, constraints, (), 1, $n$, $q$)        ▷ Compute CAD for untiled system.
    Let $r \sim 0, \sim \in \{>, \geq\}$ be the constraints bounding the index space, calculated from cad
    for all cons $\in r$ do
        for $i = 1 : n$ do
            Let $c$ be the tiled version of cons.
            Let $u$ be the subterm of $c$ including $o_i$.
            if $\operatorname{sgn}(u) \geq 0$ then
                replace $o_i$ by $o_{i,\max}$
            else
                replace $o_i$ by $o_{i,\min}$
            end if
        end for
    end for
end procedure
```

Algorithm 3 Overapproximation 1

Input: K, p, o - matrices and vectors according to Equation 2.1

Output: Overapproximation of the tile described by the input parameters

```
function OVERAPPROXTILE($o, K, p$)
    $o' \leftarrow \emptyset$
    for all $o_i \in o$ do
        while $o_i$ depends on some $o_j, j > i$ do
            $o'_i \leftarrow \left(\left(o - K^{-1}o\right)_i, \left(o - K^{-1}o + p\right)_i\right)$
        end while
    end for
    return $o'$
end function
```

Theorem 3.1. *The Extreme Value Algorithm in Algorithm 2 on the preceding page using the overapproximation shown in Algorithm 3 on the previous page is correct i.e. it computes a superset of the original index set.*

Proof. Let P be the polyhedron describing the original tiled index set, and let P' be the polyhedron after applying our algorithm to P. Then these polyhedra can be expressed via

$$P = \{x \in \mathbb{R}^n | x = Lt + o \wedge M(Lt + o) \geq q\}, \quad P' = \{x \in \mathbb{R}^n | x = Lt + o' \wedge M(Lt + o') \geq q\},$$

where M, q follow Definition 2.4 on page 3, L, t, o follow the definition of parallelepiped parametric tiling given in Section 2.2.3 and o' is the result of our overapproximation.
In order to show the correctness of our algorithm, it suffices to show $P \subseteq P'$.
Let $x \in P$ be any arbitrary point in the polyhedron P and let t, o be points such that $x = Lt + o$. For convenience, let $x' = Lt + o'$ be the point x after applying our overapproximation. Further, let $j = 1, \ldots, n$ be arbitrary.
Then we know

$$q_j \leq (Mx)_j = (M(Lt + o))_j = (MLt)_j + (Mo)_j = (MLt)_j + \sum_{k=1}^{n} m_{jk} o_k.$$

Now, according to our algorithm we distinguish between two cases for any $k = 1, \ldots, n$:
Case 1: $m_{jk} \geq 0$:
By our algorithm, we substitute o_k by $o'_k := \left(o - K^{-1}o + p\right)_k$. Using this, we get

$$\begin{aligned}(Mx')_j &= \sum_{k=1}^{n} \left(m_{jk}(Lt)_k + m_{jk}o'_k\right) \\ &= \sum_{k=1}^{n} \left(m_{jk}(Lt)_k + m_{jk}o_k - m_{jk}(K^{-1}o)_k + m_{jk}p_k\right) \\ &= (Mx)_j + \sum_{k=1}^{n} m_{jk} \left(p_k - (K^{-1}o)_k\right)\end{aligned}$$

Through the definition of parametric parallelepiped tiling, we know that $K^{-1}o \leq p$ i.e. that $\left(p_k - (K^{-1}o)_k\right) \geq 0$. This implies that $\sum_{k=1}^{n} \left(p_k - (K^{-1}o)_k\right) \geq 0$.
Using this knowledge, we get that $q_j \leq (Mx)_j \leq (Mx')_j$, hence we know that $x \in P'$.
Case 2: $m_{jk} < 0$:
By our algorithm, we substitute o_k by $o'_k := \left(o - K^{-1}o\right)_k$. Using this, we get

$$\begin{aligned}(Mx')_j &= \sum_{k=1}^{n} \left(m_{jk}(Lt)_k + m_{jk}o'_k\right) \\ &= \sum_{k=1}^{n} \left(m_{jk}(Lt)_k + m_{jk}o_k - m_{jk}(K^{-1}o)_k\right) \\ &= (Mx)_j + \sum_{k=1}^{n} (-\underbrace{m_{jk}}_{<0})(K^{-1}o)_k \\ &= (Mx)_j + \sum_{k=1}^{n} |m_{jk}|(K^{-1}o)_k\end{aligned}$$

Since $|m_{jk}| \geq 0$ always holds and since $(K^{-1}o)_k \geq 0$ holds through the definition of our tiling, this implies that $\sum_{k=1}^{n} |m_{jk}|(K^{-1}o)_k \geq 0$ holds.

This means that $q_j \leq (Mx)_j \leq (Mx')_j$, hence we have that $x \in P'$.

Concluding, we showed that for all dimensions j and any arbitrary x in any polyhedron P, our overapproximation yields a superset P' over the original P, therefore our algorithm is correct. □

As a caveat, this algorithm will only terminate if K is triagonal. This problem occurs if we allow arbitrary parallelepipeds as tile shapes, e.g. in tiles whose shape is defined by

$$K = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}.$$

The vectors defining the shape are clearly linearly independent. If one computes the tiled system using this matrix, each variable o_1, o_2 will depend on the other one in both the lower and upper limits. In such cases, our algorithm will not terminate.

But we still can prove that it does terminate under the stated condition.

Theorem 3.2. *Let $n \in \mathbb{N}$. Let $K \in \mathbb{Q}^{n\times n}$ denote the matrix defining the shape of the tile.*

Then the Extreme Value Algorithm in Algorithm 2 on page 29 using the overapproximation shown in Algorithm 3 on page 29 terminates if K is triangular.

Proof. W.l.o.g. assume that K is an upper triangular matrix. Then by Lemma A.1 on page 57 we know that K^{-1} is an upper triangular matrix again, i.e. it has the shape

$$K^{-1} = \begin{bmatrix} k'_{11} & \dots & \dots & & k'_{1n} \\ 0 & k'_{22} & \dots & & k'_{2n} \\ \vdots & 0 & \ddots & & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & k'_{nn} \end{bmatrix}.$$

Let $j = 1, \dots, n$ be arbitrary.

Since K^{-1} is upper triangular, we know that the equality

$$\sum_{\substack{i=1 \\ i\neq j}}^{n} k'_{ji}o_i = \sum_{i=j+1}^{n} k'_{ji}o_i \tag{3.1}$$

holds. By the definition of parametric parallelepiped tiling we know that the equivalences

$$
\begin{aligned}
&0 \le (K^{-1}o)_j \le p_j - 1 \\
\Leftrightarrow &0 \le \sum_{i=1}^{n} k'_{ji} o_i \le p_j - 1 \\
\Leftrightarrow &0 \le k'_{jj} o_j + \sum_{\substack{i=1 \\ i \neq j}}^{n} k'_{ji} o_i \le p_j - 1 \\
\overset{(3.1)}{\Leftrightarrow} &0 \le k'_{jj} o_j + \sum_{i=j+1}^{n} k'_{ji} o_i \le p_j - 1 \\
\Leftrightarrow &- \sum_{i=j+1}^{n} k'_{ji} o_i \le k'_{jj} o_j \le - \sum_{i=j+1}^{n} k'_{ji} o_i + p_j - 1
\end{aligned}
$$

hold for tile size parameters $p_1, \ldots, p_n$.

This means that every point loop iterator o_j only depends (possibly) on the iterators $o_{j+1}, \ldots, n$. Therefore, after n elimination steps, all variables o_j only depend on tile size parameters, and the elimination process stops, hence Algorithm 3 on page 29 terminates. Thus, Algorithm 2 on page 29 terminates too, since each polynomial can only contain a finite number of point loop variables, which are replaced without introducing other point loop variables. □

Remark: In case this overapproximation can be applied, it yields the exact same result as if a Fourier-Motzkin elimination would be applied followed by simplification. But compared to Fourier-Motzkin, this method is more efficient in general (cf. Section 3.8.6).

3.8.3. Example

In this section we provide a simple example where the overapproximation we just introduced can be applied and show its results.

Consider a parametric parallelepiped tiled system consisting of a triangular index space and parallelogram shaped tiles of width w and height h. Before calculating the new constraints, we use the algorithm for computing a Cylindrical Algebraic Decomposition for computing the projections of the variables and hence eliminating the dependency of y on x, resulting in

$$0 \le x \le n, \quad 0 \le y \le n.$$

The tiled system based on this result can be described by the system (3.2)

$$
\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \ge \begin{bmatrix} 0 \\ 0 \\ -n \\ -n \end{bmatrix}, \qquad \begin{bmatrix} 0 \\ 0 \end{bmatrix} \le \begin{bmatrix} o_1 - o_2 \\ o_2 \end{bmatrix} \le \begin{bmatrix} w-1 \\ h-1 \end{bmatrix}, \qquad \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} w & h \\ 0 & h \end{bmatrix} \begin{bmatrix} t_1 \\ t_2 \end{bmatrix} + \begin{bmatrix} o_1 \\ o_2 \end{bmatrix} \tag{3.2}
$$

or, equivalently, through the inequalities

$$\begin{aligned}
0 \leq wt_1 + ht_2 + o_1 &\leq n \\
0 \leq ht_2 + o_2 &\leq n \\
o_2 \leq o_1 &\leq o_2 + w - 1 \\
0 \leq o_2 &\leq h - 1
\end{aligned}$$

As the shape defining matrix K is triangular, we can apply our algorithm here. For simplicity, we already gave the lower and upper limits of each point loop variable $o_i, i = 1, 2$ in the above system of inequalities.

As stated, we replace every occurence of such a point loop variable o_i in the inequalities involving t_i by its maximum value $o_{i,\max}$ if it occurs positively in its left-hand side, or by its minimum value $o_{i,\min}$ otherwise. For example, consider the inequation $0 \leq wt_1 + ht_2 + o_1$. According to our algorithm, since o_1 occurs positively in this constraint, we replace it by its upper limit $o_{1,\max} = o_2 + w - 1$. The result is $1 - w \leq wt_1 + ht_2 + o_2$. Now we replace o_2 by its upper limit ($o_{2,\max} = h - 1$) yielding the final result $2 - w - h \leq wt_1 + ht_2$.

The result of applying Algorithm 2 on page 29 to the entire system results in the following new system of inequatlities:

$$\begin{array}{lclcl}
2 - w - h & \leq & wt_1 + ht_2 & \leq & n \\
1 - h & \leq & ht_2 & \leq & n
\end{array}$$

As we see, the constraints do not longer contain any point loop iterators, so they are easier to check for violation with partial information. Please note that the overapproximation of a single tile is not included in the additional constraints, as the constraints on loop point variables only depend on other o_i and on parameters. So, constraint checking on loop point iterators is rather easy with partial information and will be done by our other checks if the test point can lie in any domain.

3.8.4. Refined overapproximation

As shown our overapproximation of a single tile is correct. But it only terminates if the defining matrix K is at triagonal. In order to overcome this limitation we derived our second overapproximation algorithm shown in Algorithm 4 on the next page.

Again, we have to prove that this algorithm behaves correct.

Theorem 3.3. *Let $n \in \mathbb{N}$. Let $K \in \mathbb{Q}^{n \times n}$ and $p = (p_1 - 1, \ldots, p_n - 1) \in \mathbb{Z}^n$ be the matrix and parameter vector defining our tiling system. Then the Extreme Value Algorithm depicted in Algorithm 2 on page 29 using the overapproximation shown in Algorithm 4 on the following page is correct i.e. it generates a superset of the original index set.*

Proof. Let $i = 1, \ldots, n$ denote any arbitrary row of the matrix K. It suffices to show that Algorithm 4 on the next page generates an overapproximation for a tile described by K and p in every dimension. The correctness of the combination of both algorithms then follows immediately from the correctness of the overapproximation of a single tile and our elimination procedure similar to the previous theorem.

Let

Algorithm 4 Overapproximation 2

```
function OVERAPPROXTILE($o, K, p$)                    ▷ $o, K, p$ - See Algorithm 3 on page 29
    $o' \leftarrow \emptyset$
    $K' \leftarrow K \cdot p$
    $M \leftarrow K \cdot \text{diag}(p_1, \ldots, p_n)$
    for row = 1 : $n$ do
        if SIGN($K_{\text{row},1}$) = ... = SIGN($K_{\text{row},n}$) then
            if SIGN($K_{\text{row},1} < 0$) then
                $o'_{\text{row}} \leftarrow (K'_{\text{row}}, 0)$
            else
                $o'_{\text{row}} \leftarrow (0, K'_{\text{row}})$
            end if
        else
            $o'_{\text{row}} \leftarrow$ SEPARATELIMITS($K, M$,row)
        end if
    end for
end function

function SIGN($x$)
    if $x < 0$ then return $-1$ else return 1 end if
end function

function SEPARATELIMITS($K, M$,row)
    lower $\leftarrow 0$                                   ▷ 0 - polynomial
    upper $\leftarrow 0$
    for col = 1 : $n$ do
        if $K_{\text{row,col}} < 0$ then
            lower += $M_{\text{row,col}}$
        else
            upper += $M_{\text{row,col}}$
        end if
    end for
    return (lower, upper)
end function
```

$$\operatorname{sign}\colon \mathbb{Q} \to \{-1, 1\},\ x \mapsto \begin{cases} -1 & , \text{ if } x < 0 \\ 1 & , \text{ otherwise} \end{cases}$$

be our function determining the sign of a number. Note that the case $\operatorname{sign}(x) = 0$ *cannot* happen for *all* elements in a single row of K, because K then would be singular.

As forced by our algorithm, we distinguish between two cases.

Case 1:

For this case, we assume that the signs of all elements in this row are equal i.e. that $\operatorname{sign}(K_{i1}) = \ldots = \operatorname{sign}(K_{in})$. W.l.o.g. assume that $\operatorname{sign}(K_{i1}) = 1$.

Now let $o = (o_1, \ldots, o_n)^T \in \mathbb{Z}^n$ be any arbitrary point in the tile defined by K and p. As such, it obeys

$$0 \leq (K^{-1}o)_i \leq p_i.$$

Following our algorithm, we compute

$$K'_i = \sum_{j=1}^{n} K_{ij} p_j.$$

To prove our claim, we need to show that the range $[0, K'_i]$ is a superset of the range defined by the tile, i.e. we have to show that $0 \leq 0$ and $p_i \leq K'_i$ hold. The first part holds trivially.

By our assumption, we know that $K_{ij} \geq 0, j = 1, \ldots, n$, and thus

$$K'_i = \sum_{j=1}^{n} K_{ij} p_j = K_{ii} p_i + \sum_{\substack{j=1 \\ j \neq i}}^{n} \underbrace{K_{ij}}_{\geq 0} \underbrace{p_j}_{\geq 0} \geq K_{ii} p_i. \tag{3.3}$$

Now, if $K_{ii} = 0$, then only the origin of the tile in this dimension is included in the tile, what is fulfilled by (3.3). If we would have $K_{ii} \in (0, 1)$, then again only the origin would be included because of the integrality, and the previous case applies.

Therefore, we now may assume that $K_{ii} \geq 1$. This yields

$$K'_i \overset{(3.3)}{\geq} \underbrace{K_{ii}}_{\geq 1} p_i \geq p_i,$$

which is exactly our claim.

A similar argumentation can be made for the other case, where the roles of K'_i and 0 swap.

Case 2:

In this second case, we assume that not all signs according to our sign function are equal. Following our algorithm, we compute each entry of the matrix M using

$$M_{ij} = K_{ij} p_j, \quad j = 1, \ldots, n.$$

After the algorithm SEPARATELIMITS finishes, we get the results

$$\begin{aligned}
\text{lower}_i &= \sum_{\substack{j \in \mathcal{J}(n) \\ 1 \le j \le n}} M_{ij} = \sum_{\substack{j \in \mathcal{J}(n) \\ 1 \le j \le n}} K_{ij} p_j \\
\text{upper}_i &= \sum_{\substack{j \notin \mathcal{J}(n) \\ 1 \le j \le n}} M_{ij} = \sum_{\substack{j \notin \mathcal{J}(n) \\ 1 \le j \le n}} K_{ij} p_j
\end{aligned}$$

for some index set $\mathcal{J}(n) \subseteq \{1, \ldots, n\}$ selecting all the negative entries in the i-th row of K. In other words, we have $K_{ij} < 0$ for all $j \in \mathcal{J}(n)$ and $K_{ij} \ge 0$ for all $j \in (\{1, \ldots, n\} \setminus \mathcal{J}(n))$.
As above, let $o = (o_1, \ldots, o_n)^T \in \mathbb{Z}^n$ be any arbitrary point included in the tile, as which it obeys

$$0 \le (K^{-1} o)_i \le p_i.$$

Now we have to show that $\text{lower}_i \le 0$ and $p_i \le \text{upper}_i$.
We know that

$$\text{lower}_i = \sum_{\substack{j \in \mathcal{J}(n) \\ 1 \le j \le n}} \underbrace{K_{ij}}_{<0} p_j = - \underbrace{\left(\sum_{\substack{j \in \mathcal{J}(n) \\ 1 \le j \le n}} \underbrace{|K_{ij}|}_{\ge 0} \underbrace{p_j}_{\ge 0} \right)}_{\ge 0} \le 0,$$

which proves the first part. For the second claim, we may assume that $i \notin \mathcal{J}(n)$, since otherwise the other case applies. Then we get

$$\text{upper}_i = K_{ii} p_i + \sum_{\substack{j \notin \mathcal{J}(n) \\ 1 \le j \le n \\ j \ne i}} K_{ij} p_i \overset{\text{all } K_{ij} \ge 0}{\ge} K_{ii} p_i.$$

Similar to the previous case, we finally get

$$\text{upper}_i \ge K_{ii} p_i \ge p_i,$$

what proves our second claim.

Summing up, for any arbitrary dimension $i = 1, \ldots, n$ our overapproximation algorithm generates a superset including our tile fully, hence our algorithm is indeed correct. □

And now we prove the main advantage, namely that this algorithm will always terminate.

Theorem 3.4. *Then the Extreme Value Algorithm in Algorithm 2 on page 29 using the overapproximation shown in Algorithm 4 on page 34 always terminates.*

Proof. It is easy to see in Algorithm 4 on page 34 that it terminates always. This overapproximation yields constraints for each variable $o_i, i = 1, \ldots, n$ only dependent on tile size paramters. Therefore, if we eliminate such a variable in Algorithm 2 on page 29, no other point loop variables are generated. Therefore, after at most n steps the latter algorithm terminates too. □

3.8.5. Example

Similar to above, let us examine how our refined overapproximation algorithm applies to the same example given in Equation (3.2). In this example, our shape defining matrix has the form

$$K = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}.$$

As stated, we compute the vector

$$K' = \begin{bmatrix} w + h - 2 \\ h - 1 \end{bmatrix}.$$

Following our algorithm, we get the overapproximation system

$$\begin{aligned} 0 \le o_1 &\le w + h - 2 \\ 0 \le o_2 &\le h - 1 \end{aligned}$$

After substituting every occurence of a point loop variable o_i by its maximum (or minimum, respectively) similar to the example in Equation (3.2), we get the resulting set of additional constraints:

$$\begin{array}{lllll} 2 - w - h & \le & wt_1 + ht_2 & \le & n \\ 1 - h & \le & ht_2 & \le & n \end{array}$$

In this triangular case, we obtain the same results as with our specialized overapproximation.

Having this overapproximation algorithms, we can effectively substitute a set of new constraints only using tile loop iterators and parameters. Hence, many sample points can be discarded and thus the lifting tree can be pruned early, increasing the performance of our code generation.

This would be also possible with an application of the Fourier-Motzkin algorithm. However, Fourier-Motzkin generates a possibly huge amount of new constraints, with most of them being consequences of others. If l denotes the amount of constraints containing tile loop iterators t_i, our algorithms do produce exactly l new constraints with none of them being redundant [5].

The benefit of our method comes clear on a concrete example. For a three-dimensional parametric parallelepiped tiled system having 6 constraints which include tile loop variables, Fourier-Motzkin generated over 160 constraints, while our algorithms resulted in 6 constraints.

3.8.6. Algorithmic complexity

As already stated, our algorithms bring a big advantage over the common Fourier-Motzkin method. Now we want to briefly state what the computational complexity of our algorithms are.

For this, let m denote the number of constraints generated by computing a Cylindrical Algebraic Decomposition for the much smaller, untiled system and selecting the projections for each variable. Further, let n be the dimensionality of K.

Then we distinguish between our two overapproximation methods.

[5] If some would be redundant, they would have been already redundant before applying our algorithm.

Case 1: Overapproximation method 1 *is used.*
The computation of the overapproximation of a single tile based on parameters is done efficiently in $O(n(n+1)/2)$. The elimination algorithm in 2 on page 29 itself performs at most n variable eliminations for m constraints. Overall, the worst-case computational complexity of this algorithm combination is $O(\frac{n(n+1)}{2} + mn)$.

Case 2: Overapproximation method 2 *is used.*
The computation of the overapproximation of a single tile now takes $O(2n^2 + n^2)$ operations (including all matrix-vector multiplications and the SEPARATELIMITS case). As above, the elimination algorithm itself takes $O(mn)$ operations. So, overall this method requires $O(3n^2 + mn)$ operations in the worst-case.

Regardless of the method used, their respective computational complexities are much lower then the doubly exponential complexity of the Fourier-Motzkin method. Since our experiments showed that both algorithms can be computed quite efficiently in practice, we implemented the more general approach using method 2.

3.9. Eliminating Section Code Generation

As we learned in previous sections, function calls are expensive and we try to avoid them as often as possible. In the general case not much optimization is possible except for the lifting tree pruning techniques we described in Section 3.5.

3.9.1. Eliminating Intermediate Section Code Generation

However, we can improve this further for special cases which occur often in practice. One such common case is depicted in Figure 3.2. As this case shows, we have two neighboured sectors delimited by a section. In the general case, one needs to invoke the recursive computation for this section too. But, if we are in the situtation as depicted in the picture and certain properties hold, we can simply include the section in the boundaries of one of the sectors, thus eliminating the need for recursively generating code for that section.

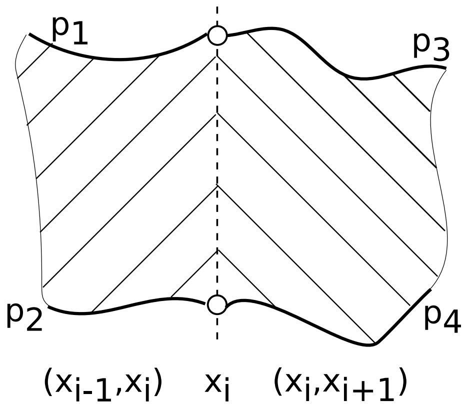


Figure 3.2.: Concept where code generation for x_i can be omitted

Many different approaches might work for describing this simplification, we choose for the following one. Informally, if multiple index sets are present, we cannot ensure a general criterion for this simplification. If we have exactly one index set, it is very hard to argue over this if the index set has "holes" in it. But, if this one index set does not have holes and the functions describing the set (evaluated at the section point) are equal, then we know that we can safely just include the section into the boundaries.

First, let us recall what a *simply connected* set is.

Definition 3.2. *Let $S \subset \mathbb{R}^n$ be any non-empty set.*

S is called simply connected *if there do not exist open sets $U, V \subset \mathbb{R}^n$ such that*

- *$S \subset U \cup V$, and*
- *$S \cap U$ and $S \cap V$ are disjoint and non-empty.*

Using this we are now able to formalize our idea from above.

Theorem 3.5. *Let $x_i \in \mathbb{A}$ be any section of the only iteration domain $\mathcal{I}$ such that both sectors (x_{i-1}, x_i) and (x_i, x_{i+1}) include points which satisfy the constraints passed to the CAD, and let $M(x, y)$ denote the set of projected polynomials not vanishing over a sector (x, y). Further, let $p_i, i = 1, \dots, 4$ be polynomials such that*

$$p_1 = \underset{p \in M(x_{i-1}, x_i)}{\arg\max}\ p(x_i), \quad p_2 = \underset{p \in M(x_{i-1}, x_i)}{\arg\min}\ p(x_i), \quad p_3 = \underset{p \in M(x_i, x_{i+1})}{\arg\max}\ p(x_i), \quad p_4 = \underset{p \in M(x_i, x_{i+1})}{\arg\min}\ p(x_i)$$

Then the code for the section x_i can be generated by including x_i into the boundary of either surrounding sector if

1. *the iteration domain $\mathcal{I}$ is simply connected according to Definition 3.2, and*
2. *p_1, p_3 exist and $p_1(x_i) = p_3(x_i)$, and*
3. *p_2, p_4 exist and $p_2(x_i) = p_4(x_i)$.*

Proof. Let $x_i \in \mathbb{A}$ denote any such section and let the constraints 1, 2, 3 hold.

Because the iteration domain is simply connected, we know that it cannot have *holes* in it, therefore if the sectors (x_{i-1}, x_i) and (x_i, x_{i+1}) participate in the decomposition, code has to be generated for the section x_i too.

Furthermore, if p_1, p_3 exist and if $p_1(x_i) = p_3(x_i)$, then we know that all points (x_i, y_j) over that section with $y_j \leq p_1(x_i)$ are part of the decomposition. That is, because the polynomials p_1, p_3 define an upper bound for both sectors and if they meet in the section, there cannot be points which belong exclusively to one sector but not to the other one.

Similarly, if p_2, p_4 exist and if $p_2(x_i) = p_4(x_i)$ holds, all (x_i, y_j) with $y_j \geq p_2(x_i)$ participate in the decomposition.

In conclusion, under the given conditions all iteration points $(x_i, p_2(x_i)) \leq (x_i, y_j) \leq (x_i, p_1(x_i))$ must be executed, which are exactly all iteration points possible over section x_i under our assumptions. Then it suffices to include the section x_i into the boundary of either sector to do so. □

Theorem 3.5 requires a simply connected set. This is a strong requirement, which cannot be inferred from the set of input polynomials. We therefore require the user to specify if this property holds. Even if this information is given, we have to evaluate polynomials at the section point, what increases the computation time. Therefore, we search for criteria which also imply that no check at the section is required.

One criterion we found is the convexity of index sets. The following lemma proves that convex sets fulfill the above criterion.

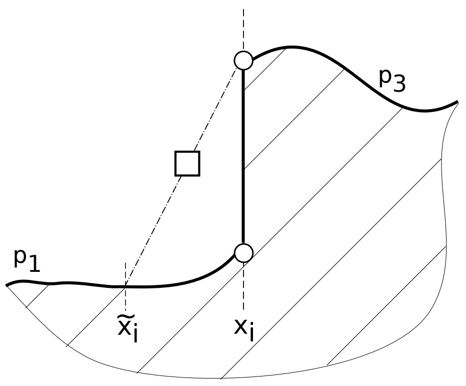


Figure 3.3.: Sketch for the proof of Lemma 3.2

Lemma 3.2. *Let $\mathcal{I}$ be a convex iteration domain.*
Then $\mathcal{I}$ supports the simplification described in Theorem 3.5 on the preceding page.

Proof. Suppose any section $x_i \in \mathbb{A}$ like in the previous theorem and let p_1, p_3 (or p_2, p_4 respectively) be the upper bounds (if existing) of the surrounding sectors. We show that $p_1(x_i) = p_3(x_i)$ in this case.
We assume for contradiction that $p_1(x_i) \neq p_3(x_i)$ and w.l.o.g. we assume that $p_3(x_i) > p_1(x_i)$. Further, define $\widetilde{x_i} = x_i - \varepsilon$ for a small enough $\varepsilon > 0$. In this small environment we may assume that p_1 is monotone. Figure 3.3 illustrates this. Additionally, define $\square = (\square.x, \square.y) = (\alpha\widetilde{x_i} + (1-\alpha)x_i,\ \alpha p_1(\widetilde{x_i}) + (1-\alpha)p_3(x_i))^T$ to be the convex combination between the two depicted points with $\alpha \in (0,1)$.
Now we distinguish between two cases.
Case 1: p_1 is monotone rising in $[x_i - \varepsilon, x_i]$.
Since $p_3(x_i) > p_1(x_i)$ we are able to find a value for α close enough to 1 such that $\square.y > p_1(\square.x)$ i.e. $\square \notin \mathcal{I}$, what contradicts the convexity property.
Case 2: p_1 is monotone falling in $[x_i - \varepsilon, x_i]$.
In this case, we know that $\square.y > p_1(x_i - \varepsilon) \geq p_1(\square.x)$ for all α, hence $\square \notin \mathcal{I}$, what contradicts the convexity property.
So in all cases we have $\square \notin \mathcal{I}$, but this contradicts the convexity of the index set. Hence our assumption was false and $p_1(x_i) = p_3(x_i)$ holds.

A very similar argument can be made to show $p_2(x_i) = p_4(x_i)$ if these polynomials exist, and since it is a well-known fact that every convex set is simply connected all preconditions of Theorem 3.5 on the preceding page are fulfilled. □

However, as it turns out this cannot be inferred from the set of input polynomials either.
To cope with this problem, we focussed on the problem of polyhedral constraints, as they occur very often. Lemma B.1 on page 57 in Appendix B proves that in fact all polyhedra are convex sets. And since each polyhedron is described by a set of affine-linear polynomials, we now gained an inferrable criterion when our optimization holds.
Note that in general the set inclusion hierarchy

$$\text{polyhedral} \subset \text{convex} \subset \text{simply connected}$$

holds.

3.9.2. Elimination of bounding sections code generation

Similarly to above, we can define a criterion for when code has to be generated explicitly for the first and last sections in a certain dimension i.e. for the sections x_1, x_n such that their predecessor (successor) is the sector $(-\infty, x_1)$ (or $(x_n, +\infty)$ respectively).

As such, if we have a simply connected set and if all constraints use only $\geq$ or $\leq$, we can include the sections into the upper (or lower, respectively) boundary of the generated code without recursively generating it for these sections.

3.9.3. Implementation

In our implementation, we decided to follow the described principles as follows.

- Our simplification is guaranteed to work only if exactly one index set is in play.
- The user has to specify whether the index set can be assumed to be simply connected. Furthermore, the user can specify whether the set can be assumed to be convex. If the input is polyhedral, we can infer both properties due to the above hierarchy.
- If we either inferred that the input set is convex or if the user specified it, we can immediately add the section to the boundary of one of the surrounding sectors.
- If the input set is not convex but simply connected as specified by the user, we check the requirements of Theorem 3.5 on page 39. If these requirements are fulfilled, we add the section to the sector code without computation, otherwise we have to invoke the recursive procedure for the section.
- If the respective section is either the first or last one in the current dimension, we check if they can be added without computation as described in Section 3.9.2.

Further research should be done in order to derive more general inferrable criteria describing when this optimization can be applied.

3.10. Affine Projection

As mentioned in Section 3.3.2, the choice of the projection operator has a big influence on the overall code generation performance. We also stated that we choose the projection strategy of Brown as it promises the best performance in the average case.

But as we will show, we can improve this projection further for the special case of linear-affine polynomials.

Before we do that, we have to introduce some terminology.

Definition 3.3. *Let $f = a_n x_k^n + \ldots + a_0 \in \mathbb{Q}[x_1, \ldots, x_k]$ be a polynomial with $a_n \neq 0$. Then*

$$ldcf(f) = a_n$$

is the leading coefficient *of f.*

For convenience, let us define an expression for the irreducible factor of a polynomial.

Definition 3.4. *Let $f = a_n x_k^n + \ldots + a_1 x_k + a_0 \in \mathbb{Q}[x_1, \ldots, x_k]$ be a polynomial with main variable x_k. Then we define* coeffs(f) *to be the greatest common divisor of all its coefficients, i.e.*

$$coeffs(f) = gcd(a_0, \ldots, a_n).$$

Further, we need to define the discriminant and the resultants in the context of polynomials which were originally defined in [Syl40] although not explicitly mentioned. Collins re-introduced the concept of resultants and the Sylvester matrix in [Col67].

Definition 3.5. *Let $f = a_l x_k^l + \ldots + a_1 x_k + a_0 \in \mathbb{Q}[x_1, \ldots, x_k]$ be a polynomial and let $g = b_m x_k^m + \ldots + b_1 x_k + b_0 \in \mathbb{Q}[x_1, \ldots, x_k]$ be another polynomial.*
The Sylvester matrix *[Col67] $S(f,g)$ of f and g is a $(l+m) \times (l+m)$ matrix defined by*

$$S(f,g) = \begin{bmatrix} a_l & a_{l-1} & \ldots & a_0 & & & \\ & a_l & a_{l-1} & \ldots & a_0 & & \\ & & \ddots & \ddots & & \ddots & \\ & & & a_l & a_{l-1} & \ldots & a_0 \\ b_m & b_{m-1} & \ldots & b_0 & & & \\ & b_m & b_{m-1} & \ldots & b_0 & & \\ & & \ddots & \ddots & & \ddots & \\ & & & b_m & b_{m-1} & \ldots & b_0 \end{bmatrix}$$

The resultant res(f,g) *of f and g is given through the determinant* det $S(f,g)$ *[Col67].*

Definition 3.6. *Let $f = a_n x_k^n + \ldots + a_1 x_k + a_0 \in \mathbb{Q}[x_1, \ldots, x_k]$ be a polynomial of degree n. The* discriminant discr(f) *of f is given through [GKZ09]*

$$discr(f) = (-1)^{n(n-1)/2} a_n^{2n-2} \prod_{i<j} (r_i - r_j)^2,$$

where $r_1, \ldots, r_n$ are the roots of f.

Finally, we can state Browns projection operator and an optimization for the affine case.
In his paper, Brown defined his projection strategy of a set of polynomials A to be the union of the set of all leading coefficients, the set of all discriminants of polynomials f and the set of all resultants of pairs f, g of distinct elements of A [Bro01].
In mathematical notation, this can be rewritten as follows.

Definition 3.7. *Let A be a squarefree basis in $\mathbb{Q}[x_1, \ldots, x_k], k \geq 2$.*
The projection operator projB *of Brown is defined by*

$$\begin{aligned} projB(A) &= projB_1(A) \cup projB_2(A) \\ projB_1(A) &= \bigcup_{f \in A} \{discr(f)\} \cup \{ldcf(f)\} \cup \{coeffs(f)\} \\ projB_2(A) &= \bigcup_{\substack{f,g \in A \\ f \neq g}} \{res(f,g)\} \end{aligned}$$

A common case which occurs often in practice is that the polynomials are affine w.r.t. the currently processed variable x_k. In this case, we can minimize the number of projected polynomials further to our affine projection operator.

Definition 3.8. *Let A be a set of affine-linear polynomials in $\mathbb{Q}[x_1, \ldots, x_k], k \geq 2$. Define*

$$\begin{aligned} projAff(A) &= projAff_1(A) \cup projB_2(A) \\ projAff_1(A) &= \bigcup_{f \in A} \{ldcf(f)\} \cup \{coeffs(f)\} \end{aligned}$$

Naturally, we have to show that this projection operator performs correctly i.e. that it is delineating. We do so by showing the equivalence to Browns operator.

Lemma 3.3. *Let A be a set of polynomials in* $\mathbb{Q}[x_1, \ldots, x_k]$ *which are affine w.r.t. the variable* x_k*. Then* projAff*(A) produces the same set of polynomials as* projB*(A).*

Proof. The only difference between these two operators is that the set of discriminants was omitted in our own projection. Therefore we show that these discriminants are not relevant for projection.

Let $f = \alpha x_k + \beta \in \mathbb{Q}[x_1, \ldots, x_k]$ be an affine-linear polynomial. Since this polynomial is affine-linear w.r.t. x_k, it has at most one root. But then the discriminant does not exist (since it would require two distinct roots) and it is commonly defined to be 1, which is constant and therefore not relevant for projection. □

As we just showed, our projection is equivalent to Browns for the special case of affine polynomials. Because of this, the set of projected polynomials is identical to the set projected by Browns operator, as the latter computes the discriminant but discards it afterwards.

Since we do not compute the discriminant, we reduce the computational effort and hence we expect a little improvement of the overall code generation time. This is confirmed by our experiments in Chapter 4.

Chapter 4

Experiments

After describing all needed prerequisites and our implementation including the applied optimizations, this chapter presents the performance evaluation of this implementation.

4.1. Experimental Setup

First, let us describe the setup on which we tested our system.

4.1.1. Resources

In our experiments, we used both the computers *aesche* and *nervling* with each having four cores supporting hyperthreading, making eight virtual cores in total. They use an Intel® Core™ *i*7−4790 CPU clocked at 3.6 GHz.

On the software side, we used the freely available ***C**omputer **A**rithmetic **L**ibrary* [1] at revision `e1c17fa5f5713c4ff0223fbc5024655d91ac8cbe`, which is a prerequisite of the ***S**atisfiability-**M**odulo-**T**heories **R**eal **A**lgebra **T**oolbox* [2] (SMT-RAT) at revision `6af593b0d51ae2849774617432baed8c31364eae`. Additionally, the `SMT-RAT` library requires the *GNU Multiple Precision Arithmetic Library*, for which we use version 10.3.0.

To build these libraries and our own implementation, we choose `gcc` as compiler and we used `gcc` 5.4.0. For each of the above libraries, we specified to use a thread-safe build and a release build. Our application itself can be built by invoking

```
g++ -Wall -std=c++14 -O3 -I<paths/to/includes> -L<paths/to/libraries> \
    smtrat_tests.cc                                                 \
    -ltcmalloc -lcarl -lsmtrat -lgmpxx -lgmp -lpthread
```

4.1.2. Test cases

For our experiments, we distinguish between the nine test cases *rect-rect, rect-par, rect-invpar, tri-rect, tri-invpar, nonconvex, nonconvex2, sphere* and *steinmetz*, for each of which we measured the execution time of the code generation phase and took the average over ten runs.

The first five test cases are rectangular/triangular index sets with different parametric tilings applied. Each tiling consists of two parameters for a tiles width and height. Table 4.1 on the next page summarizes this.

For each of these test cases, we compute the parametric tiling on our own, where we use the tile shape defining matrices

$$K_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad K_2 = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \quad K_3 = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$$

for rectangular (K_1), paralleogram (K_2) and inverse parallelogram (K_3) tilings.

[1] https://github.com/smtrat/carl

[2] https://github.com/smtrat/smtrat

Index set	Description	Tiling	Abbreviation
rectangular	$\{(x,y) \in \mathbb{Z}^2 \mid 1 \le x \le 12 \wedge 1 \le y \le 6\}$	rectangular	rect-rect
		parallelogram	rect-par
		inverse parallelogram	rect-invpar
triangular	$\{(x,y) \in \mathbb{Z}^2 \mid 0 \le x \wedge 0 \le y \le p - x\}$	rectangular	tri-rect
		inverse parallelogram	tri-invpar

Table 4.1.: Test cases involving tiling

It is easy to see that the test cases are increasing in complexity, especially with the triangular case involving an additional parameter in the index set description. Considering the doubly-exponential nature of the Cylindrical Algebraic Decomposition, we expect a significant difference in execution times.

The remaining test cases include non-polyhedral constraints. The first test case (*nonconvex*) uses a non-convex index set described by

$$D_1 = \{(x,y) \in \mathbb{Z}^2 \mid 10 \le x \le 70 \wedge 10 \le y \le 90 \wedge (y-40)^2 - 30x + 1200 \ge 0 \wedge (x-70)^2 + (y-80)^2 - 100 \ge 0\}.$$

Quite similar, the test case *nonconvex2* is described via the index set

$$D_2 = \{(x,y) \in \mathbb{Z}^2 \mid 1 \le x \le 7 \wedge 1 \le y \le 9 \wedge (y-4)^2 - 3x + 12 \ge 0\}.$$

Finally, the test cases *sphere* and *steinmetz* describe the three-dimensional sphere of radius 2 and the Steinmetz solid, respectively, and are defined by

$$\begin{aligned} D_3 &= \{(x,y,z) \in \mathbb{Z}^3 \mid x^2 + y^2 + z^2 \le 4\}, \\ D_4 &= \{(x,y,z) \in \mathbb{Z}^3 \mid x^2 + y^2 \le a^2 \wedge y^2 + z^2 \le a^2 \wedge x^2 + z^2 \le a^2\}, \quad a \in \mathbb{Z}, \end{aligned}$$

where a is a parameter.

4.2. Improvements over the basic implementation

In this part, we compare the basic implemented code generation procedure successively to the implementation with optimizations activated.

In our basic implementation we noticed that execution times varied from milliseconds (test case *nonconvex*) to nearly three hours (test case *tri-invpar*).

In order to improve the code generation speed, we applied our optimizations described in Chapter 3 one after another to our procedure. Table 4.2 on the facing page shows the execution times of our polyhedral test cases at all optimizations. The optimizations are applied incrementally and stay activated for all later tests. The only exception is the parallelized version, which uses all previous optimizations, but the version including the *Simplification* optimization is not executed in parallel. For each of these tests we used Brown's projection operator.

Similarly, Table 4.3 on the next page shows the same order of optimizations activated for our non-polyhedral test cases. Execution times are now given in milliseconds.

For brevity, the optimized memory allocation will be referred as *Memory*, *Loop* indicates that our loop fusion technique is activated, *Early* denotes the usage of our tree pruning techniques, *Overapp.* denotes our tiling specific overapproximation, and *Simpl.* indicates that our section code elimination simplification was used if possible.

Test	Original	Memory	Early	Loop	Overapp.	Simpl.	Parallel
rect-rect	9.64	7.61	1.44	1.45	1.39	0.05	1.97
rect-par	1,092	869	67	69	55	2	59
rect-invpar	3,122	2,383	161	164	168	6	177
tri-rect	3,703	2,914	365	377	144	9	180
tri-invpar	10,757	8,644	1,024	1,027	679	45	814

Table 4.2.: Execution times including optimizations (seconds, polyhedral tests)

Test	Original	Memory	Early	Loop	Overapp.	Simpl.	Parallel
nonconvex	18.0	14.1	10.6	10.0	10.2	5.7	10.4
nonconvex2	116.6	102.4	93.0	93.8	98.6	22.3	93.8
sphere	9.7	6.7	7.0	7.6	7.6	3.5	7.8
steinmetz	462.8	382.9	261.8	260.9	263.0	80.5	280.3

Table 4.3.: Execution times including optimizations (milliseconds, non-polyhedral tests)

Figure 4.1 depicts the total speedup of our optimizations compared to our original basic implementation as derived from Table 4.2 for the polyhedral test cases. As described, the execution times (and therefore the speedups) were measured incrementally, i.e. preceeding optimizations stay activated for later measurements. As an example, the optimization called *Early Return* was active for the last three measurements.

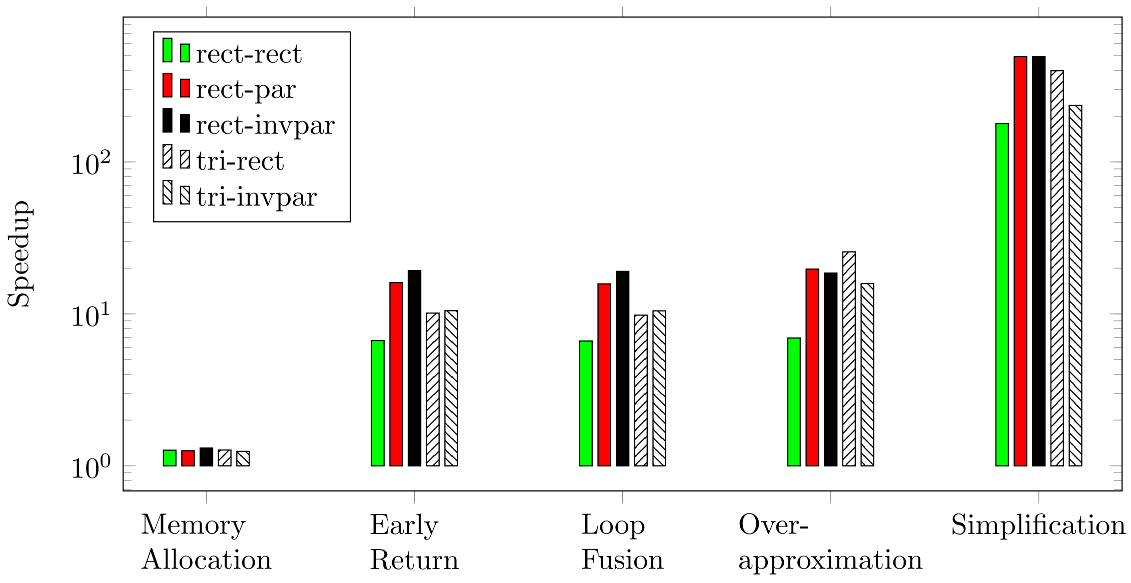


Figure 4.1.: Total speedup over basic implementation (polyhedral tests)

As one can see, our optimizations are beneficial especially for higher-dimensional systems created by tiling the index space.

The first big speedup is reached with the optimizations of pruning the lifting tree and by using additional constraints computed for tiled systems with our overapproximation.

The graph shows that the loop fusion did not increase performance very much as expected. However, it greatly reduces memory consumption by a factor of about 300 bytes for every loop fused into another one, which is essential for dealing with more complex code generation problems.

In the more complicated test cases we notice, that our overapproximation (which yields additional constraints on tiled system), our application gets about 2.6 times as fast as without it [3].

Clearly, the most beneficial optimization is the simplification by eliminating section code generation as described in Section 3.9, which yields total speedups of up to 500 times faster than the original implementation. However, even though this optimization is applicable for many cases in practice, it is not for all.

Sadly, parallelism did not bring any benefit in our application due to the excessive use of pooling datastructures in the library `SMT-RAT`. In fact, the performance decreases a little bit because of synchronization overhead. We hope that future work will compensate this disadvantage.

Performing the same experiments for our non-polyhedral test cases, we obtain the graph shown in Figure 4.2

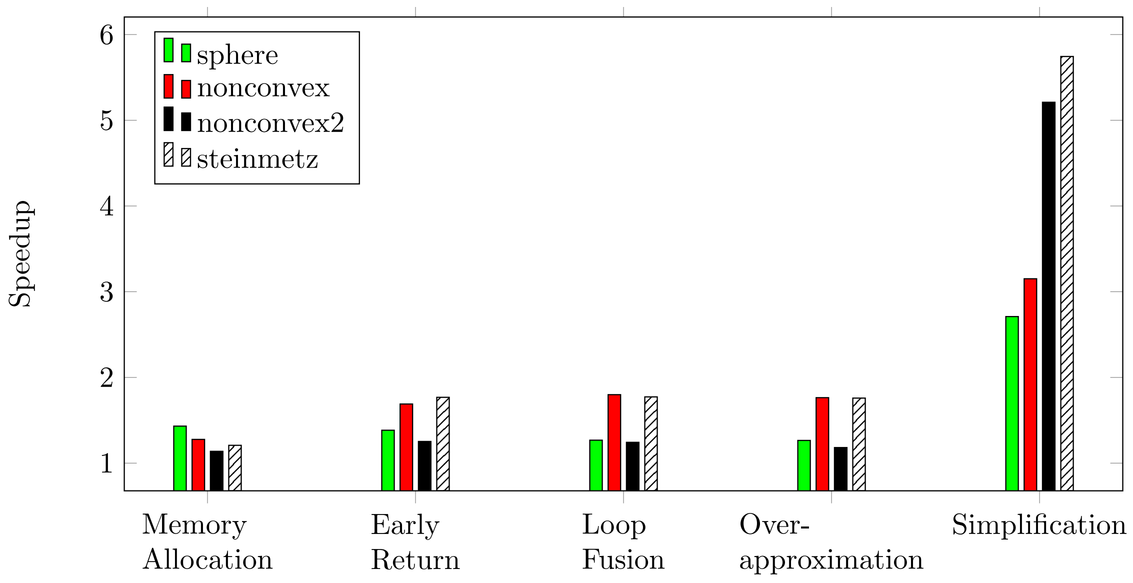


Figure 4.2.: Total speedup over basic implementation (non-polyhedral tests)

Clearly, memory allocation does not yield a big improvement as there is little memory to transfer here due to the small workload induced by the test cases.

We see further that the lifting tree pruning optimization improves our non-polyhedral cases even further, and we expect this optimization to become more important in bigger test cases like the high-dimensional systems shown earlier.

As expected, the loop fusion has no impact on the execution times, but again this will become more important for bigger systems.

Further, the overapproximation did contribute nothing to the performance. But this is also clear, as the overapproximation can be used for tiled systems only, and we did not tile our index sets here.

Finally, just like in the polyhedral case, the function call elimination optimization did bring some improvement, although not as big as before, for the simple reason that there are fewer function calls to eliminate.

[3]Note that Figure 4.1 on the preceding page uses a logarithmic scale.

Similar to the shown tests, we did the same experiments as above but using our own affine projection operator as described in Section 3.10 where applicable. The results are shown in Table 4.4.

Test	Original	Memory	Early	Loop	Overapp.	Simpl.	Parallel
rect-rect	9.84	7.66	1.46	1.48	1.38	0.06	1.93
rect-par	1, 082	858	67	68	55	2	60
rect-invpar	3, 033	2, 432	162	167	169	6	176
tri-rect	1, 492	1, 214	123	126	36	2	44
tri-invpar	2, 403	1, 808	197	196	117	9	138

Table 4.4.: Execution times using our affine projection

When comparing the above results to our previous ones using Brown's projection operator, we see that the improvements compared optimization-by-optimization are roughly the same over all optimizations. This indeed indicates that although our projection is equivalent to Brown's, the computation of the (in this case) obsolete discriminant takes a long time which can be omitted safely. As Table 4.4 clearly shows, this optimization improves the code generation speed especially in the triangular test cases, where e.g. for the test case *tri-invpar* the runtime drops by a factor of 5 from 45 to 9 seconds.

Fortunately, these improvements become more evident for more complex test cases like the tiled triangular index space. This hints that our affine projection operator performs better in the special case of parametric parallelepiped tiling than Brown's projection operator.

Independence of Optimizations

Additionally, we also tested our optimizations for both projection strategies individually activated. The goal of this experiment was to determine whether our optimizations are independent of each other, for which we compared the overall speedup of each test case according to the above tables with the product of the individual speedups of the individual optimizations.

It should be noted that our *Overapp.* optimization only generates more constraints to check by the optimization *Early* but does not perform the checks itself. Therefore, the activation of *Overapp.* implies the optimization *Early*.

Table 4.5 on the next page shows the execution times of our test cases with each column being the only active optimization during the respective run. The execution times for tests marked with * are given in milliseconds, while the other times are given in seconds.

Now we compare the speedups. For example, for the testcase *tri-invpar* using our affine projection strategy, the overall speedup, i.e. the speedup of all optimizations excluding parallelism compared to the original execution time is about 259, while the product of the speedups according to Table 4.5 on the following page for this test case is 304.

The deviation between these two values is clearly the dependence of the optimization *Overapp.* on *Early* as we described above.

But besides this, our optimizations seem to perform nearly independently.

Projection	Test	Original	Memory	Early	Overapp.	Simpl.
Brown	rect-rect	9.64	7.61	1.44	1.39	13.02
	rect-par	1,092	869	67	56	1,356
	rect-invpar	3,122	2,383	161	169	3,371
	tri-rect	3,703	2,914	365	146	4,962
	tri-invpar	10,757	8,644	1,024	683	11,347
	nonconvex*	18	14	10	11	10
	nonconvex2*	116	102	93	45	94
	sphere*	9	6	7	2	8
	steinmetz*	462	382	261	244	262
Affine	rect-rect	9.84	7.66	1.46	1.36	12.9
	rect-par	1,082	858	67	55	1,319
	rect-invpar	3,033	2,432	162	169	3,422
	tri-rect	1,492	1,214	123	36	2,317
	tri-invpar	2,403	1,808	197	118	2,593

Table 4.5.: Execution times with only one optimization active at a time

4.3. Evaluation against a Haskell prototype

In this section, we evaluate the performance of our code generation procedure compared to an existing `Haskell` prototype.

For this, we implemented our test cases in the existing Haskell prototype, but we noticed that only four out of our nine test cases yielded results. The execution times of the Haskell prototype as well as our already shown times are presented in Table 4.6.

	sphere	*nonconvex*	*nonconvex2*	*rect-rect*
`C++`	0.397 [ms]	0.633 [ms]	2.391 [ms]	0.057 [s]
`Haskell`	72 [ms]	107 [ms]	348 [ms]	159.350 [s]
Speedup	181×	169×	146×	2,795×

Table 4.6.: Comparison of execution times of the Haskell prototype to our implementation

For all other test cases, the prototype either ran out of memory or we aborted the computation after several hours.

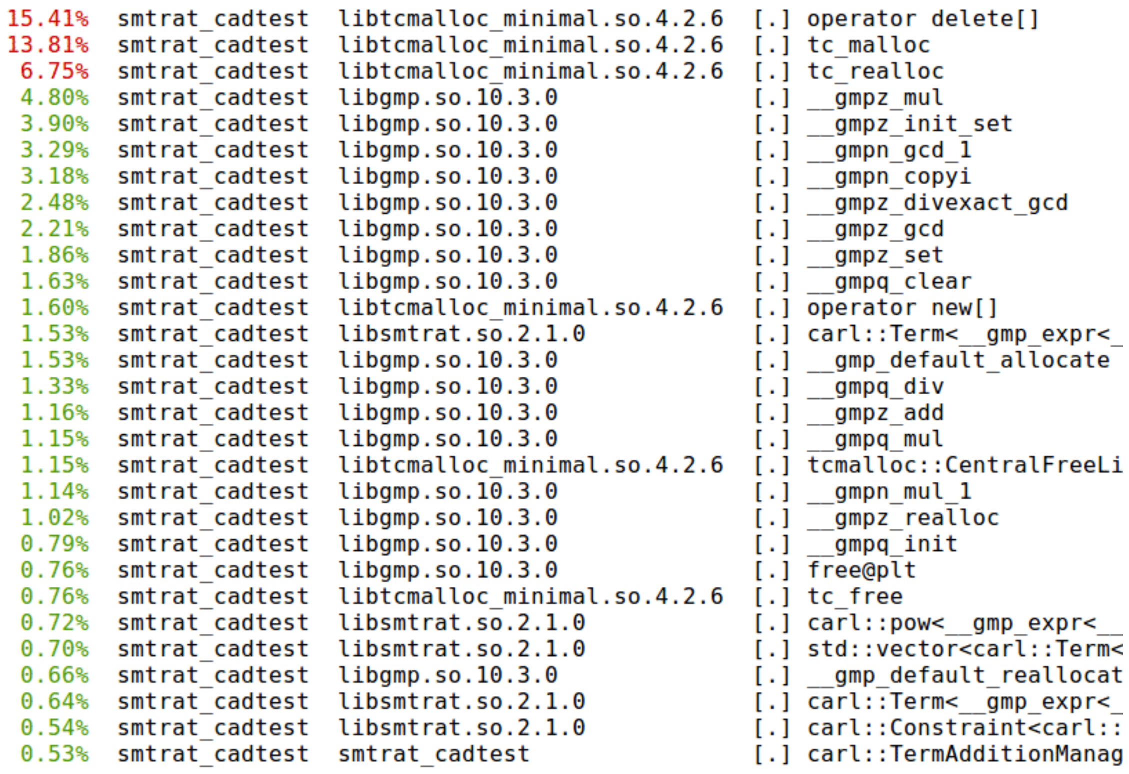
This behaviour hints that our solution is expected to scale better with higher-dimensional systems than the existing implementation. This is especially true for polyhedral cases, as the existing implementation only yielded a result for our simplest test case, while our approach barely needed any additional memory and produced a correct result in a few seconds.

Overall, we can see that our implementation is superior to the existing prototypical implementation in every case we tested.

4.4. Final Program Analysis

After we finished all our experiments, we did a final program analysis using the common performance profiling tool `perf`. The results are presented in this section.

Figure 4.3 shows the (truncated) results of a profiling run with perf on test case *rect-invpar*.



```
15.41%  smtrat_cadtest  libtcmalloc_minimal.so.4.2.6  [.] operator delete[]
13.81%  smtrat_cadtest  libtcmalloc_minimal.so.4.2.6  [.] tc_malloc
 6.75%  smtrat_cadtest  libtcmalloc_minimal.so.4.2.6  [.] tc_realloc
 4.80%  smtrat_cadtest  libgmp.so.10.3.0              [.] __gmpz_mul
 3.90%  smtrat_cadtest  libgmp.so.10.3.0              [.] __gmpz_init_set
 3.29%  smtrat_cadtest  libgmp.so.10.3.0              [.] __gmpn_gcd_1
 3.18%  smtrat_cadtest  libgmp.so.10.3.0              [.] __gmpn_copyi
 2.48%  smtrat_cadtest  libgmp.so.10.3.0              [.] __gmpz_divexact_gcd
 2.21%  smtrat_cadtest  libgmp.so.10.3.0              [.] __gmpz_gcd
 1.86%  smtrat_cadtest  libgmp.so.10.3.0              [.] __gmpz_set
 1.63%  smtrat_cadtest  libgmp.so.10.3.0              [.] __gmpq_clear
 1.60%  smtrat_cadtest  libtcmalloc_minimal.so.4.2.6  [.] operator new[]
 1.53%  smtrat_cadtest  libsmtrat.so.2.1.0            [.] carl::Term<__gmp_expr<_
 1.53%  smtrat_cadtest  libgmp.so.10.3.0              [.] __gmp_default_allocate
 1.33%  smtrat_cadtest  libgmp.so.10.3.0              [.] __gmpq_div
 1.16%  smtrat_cadtest  libgmp.so.10.3.0              [.] __gmpz_add
 1.15%  smtrat_cadtest  libgmp.so.10.3.0              [.] __gmpq_mul
 1.15%  smtrat_cadtest  libtcmalloc_minimal.so.4.2.6  [.] tcmalloc::CentralFreeLi
 1.14%  smtrat_cadtest  libgmp.so.10.3.0              [.] __gmpn_mul_1
 1.02%  smtrat_cadtest  libgmp.so.10.3.0              [.] __gmpz_realloc
 0.79%  smtrat_cadtest  libgmp.so.10.3.0              [.] __gmpq_init
 0.76%  smtrat_cadtest  libgmp.so.10.3.0              [.] free@plt
 0.76%  smtrat_cadtest  libtcmalloc_minimal.so.4.2.6  [.] tc_free
 0.72%  smtrat_cadtest  libsmtrat.so.2.1.0            [.] carl::pow<__gmp_expr<__
 0.70%  smtrat_cadtest  libsmtrat.so.2.1.0            [.] std::vector<carl::Term<
 0.66%  smtrat_cadtest  libgmp.so.10.3.0              [.] __gmp_default_reallocat
 0.64%  smtrat_cadtest  libsmtrat.so.2.1.0            [.] carl::Term<__gmp_expr<_
 0.54%  smtrat_cadtest  libsmtrat.so.2.1.0            [.] carl::Constraint<carl::
 0.53%  smtrat_cadtest  smtrat_cadtest                [.] carl::TermAdditionManag
```

Figure 4.3.: Perf performance analysis

We clearly see that the majority of the overall execution time is spent in the memory allocation part, and another bigger share in the internal functions of the GNU MP library.

Closer inspection of these parts showed that the GNU MP functions actually invoke the allocations done by `tcmalloc` and that most of these GNU MP functions themselves are invoked by polynomial arithmetic functions in the libraries SMT-RAT and CArL.

For this test case we evaluated that there are over 165 million allocations, while we are only responsible for 0.2% of these allocations.

The computationally most expensive part our own code causes is shown in the very last line of Figure 4.3, which itself is a call to the term addition manager defined in the library CArL and is used to retrieve the terms of a polynomial.

Summing these facts up we see that our own code only participates little to the overall hot spots. The biggest influence of our program on the overall performance not directly shown here is the cascading recursive structure.

Additionally, we also did an analysis using the tool `perf stat` which emits common performance measuring metrics. Again, we did the analysis for the test case *rect-invpar* and executed the tests five times to get stable results.

An excerpt of the results is shown in Listing 4.1.

```
*> perf stat -d -d -r 5 ./smtrat_cadtest > test.c
Performance counter stats for './smtrat_cadtest' (5 runs):

  56.988.029.082  instructions             #    2,44  insns per cycle
  10.784.903.756  branches                 #  687,916 M/sec
      51.194.358  branch-misses            #    0,47% of all branches
  18.906.467.399  L1-dcache-loads          # 1205,950 M/sec
      87.968.589  L1-dcache-load-misses    #    0,47% of all L1-dcache hits
      12.207.755  LLC-loads                #    0,779 M/sec
         614.000  LLC-load-misses          #    10,06% of all LL-cache hits
      66.150.675  L1-icache-load-misses    #    4,219 M/sec
  18.903.426.010  dTLB-loads               # 1205,756 M/sec
       1.562.670  dTLB-load-misses         #    0,01% of all dTLB cache hits
      13.440.865  iTLB-loads               #    0,857 M/sec
         885.187  iTLB-load-misses         #    6,59% of all iTLB cache hits
```

Listing 4.1: Analysis using perf

We clearly see that we achieve a good performance indicated by a very small number of branch and cache misses relatively to the total amount of accesses. Naturally, these metrics and especially the cache misses highly depend on the used test case and other factors active during the program execution.

All in all, we can assume that our implementation only contributes little to the remaining hot spots and performs well even when dealing with high-dimensional inputs.

Chapter 5

Conclusion and Further Work

5.1. Conclusion

In the first chapter of this thesis, we provided a general introduction to the problem of code generation for index sets described using constraints allowing arbitrary polynomials. In that chapter we especially focussed on the description of the Cylindrical Algebraic Decomposition, which is our chosen method for solving the problem of code generation.

We further discussed our implementation in the programming language `C++` including several optimizations. Considering our optimizations, we can basically distinguish between two cases, namely optimizations which can be applied to every input problem, and improvements which are tuned towards special cases which occur often in practice. While the first case covers techniques like early aborting code generation if it can be proven that the iteration domain constraints are not satisfiable with the partial information, the special case optimizations include the fully automated computation of additional constraints that must be fulfilled and the elimination of recursive function calls.

Finally, we measured the execution times of our code generation procedure and evaluated how much we improved by including our optimizations. As the results show, our own implementation needed several hours in its basic form, but the stated enhancements improved it greatly with overall speedups of up to 500× can be observed when all optimizations are activated, where the most influental optimization was without a doubt the elimination of section code generation calls which effectively omit every second recursive function call. We further compared our implementation to an already existing prototypical implementation in `Haskell` and showed that our approach outperformed it in each of our tested cases.

Summing up, we implemented a more general code generation procedure in C++ and enhanced its performance greatly. We also showed in our thesis that there are several points which can and should be improved in future work.

5.2. Future Work

As mentioned, there exist points in our implementation which should be improved further. We showed that the major performance bottleneck of our application is the memory management, therefore one should investigate this. Additionally, it might be beneficial to eliminate the recursive structure (e.g. by using trampoline functions) if possible.

Besides that, we do not expect many potential for improvement considering low-level optimizations. However, there is still room for algorithmic improvements.

Our first suggestion here is that one should investigate more general and ideally inferrable criteria when our function call elimination optimization can be applied.

Our experiments further showed that, although the algorithm can be parallelized conceptually, it did not bring any benefit due to library problems. Hence, the thread-safe build of the library SMT-RAT should be fixed. From this, we expect quite some improvement.

Finally, we want to leave a note about the idea of leaving the classical Cylindrical Algebraic Decomposition behind and move on to more advanced techniques. For this, we suggest future investigation of methods such as Truth Table-invariant Cylindrical Algebraic Decomposition [BDE+14], CAD computations using triangulations or the use of Gröbner bases if equational constraints occur often.

Appendices

A. Inverse matrices of triagonal matrices

Lemma A.1. *Let $\mathbb{K} \in \{\mathbb{Q}, \mathbb{R}\}$ and let $M \in \mathbb{K}^{n \times n}$ be an invertible, upper triangular matrix. Then M^{-1} is triangular again.*

Proof. We proof this by induction over $n \in \mathbb{N}$.
Induction base ($n = 1$): Trivial.
Inductive step ($n \mapsto n + 1$):
Let $M \in \mathbb{K}^{(n+1)\times(n+1)}$ have the shape

$$M = \begin{bmatrix} m & z^T \\ 0_n & M' \end{bmatrix},$$

where $0 \neq m \in \mathbb{K}$ is the first element, $0_n, z^T \in \mathbb{K}^n$ are vectors and $M' \in \mathbb{K}^{n\times n}$ is the lower right block matrix. Now let

$$N = \begin{bmatrix} \frac{1}{m} & -\frac{1}{m} z^T (M')^{-1} \\ 0_n & (M')^{-1} \end{bmatrix} \in \mathbb{K}^{(n+1)\times(n+1)}.$$

By our induction hypothesis, $(M')^{-1}$ is an upper triangular matrix. Then $MN = I_n$ i.e. $N = M^{-1}$ and N is clearly an upper triangular matrix again. □

B. Convex Polyhedra

Definition B.1 (Convex Set)**.** *Let $n \in \mathbb{N}$ and $S \subset \mathbb{R}^n$ be a set. S is called* convex *if for all points $x, y \in S$ and for all $\alpha \in [0, 1]$ the point $\alpha x + (1 - \alpha)y \in P$.*

Lemma B.1. *Let $P = \{x \in \mathbb{R}^n | Mx \geq q\}$ be a polyhedron according to Definiton 2.4 on page 3. Then P is a convex set according to Definition B.1.*

Proof. Let P be a polyhedron as given. Let $x, y \in P$ be any arbitrary two points in that polyhedron and let $\alpha \in [0, 1]$ be arbitrary.
W.l.o.g. we can assume that $x \neq y$, otherwise the claim holds trivially.
Let $z = \alpha x + (1 - \alpha)y$ be the convex combination of the points x and y.
Then it holds that

$$Mz = M(\alpha x + (1 - \alpha)y) = \alpha \underbrace{Mx}_{\geq q} + (1 - \alpha) \underbrace{My}_{\geq q} \geq \alpha q + (1 - \alpha)q = q$$

hence $z \in P$. □

Bibliography

[AI91] Ancourt, Corinne ; Irigoin, François: Scanning Polyhedra with DO Loops. In: *SIGPLAN Not.* 26 (1991), April, Nr. 7, 39–50. http://dx.doi.org/10.1145/109626.109631. – DOI 10.1145/109626.109631. – ISSN 0362–1340

[BDE+14] Bradford, Russell J. ; Davenport, James H. ; England, Matthew ; McCallum, Scott ; Wilson, David J.: Truth Table Invariant Cylindrical Algebraic Decomposition. In: *CoRR* abs/1401.0645 (2014). http://arxiv.org/abs/1401.0645

[Bro01] Brown, Christopher W.: Improved Projection for Cylindrical Algebraic Decomposition. In: *Journal of Symbolic Computation* 32 (2001), Nr. 5, 447 - 465. http://dx.doi.org/https://doi.org/10.1006/jsco.2001.0463. – DOI https://doi.org/10.1006/jsco.2001.0463. – ISSN 0747–7171

[Bro04] Brown, Christopher W.: Companion to the Tutorial Cylindrical Algebraic Decomposition Presented at ISSAC 2004. (2004). http://www.usna.edu/Users/cs/wcbrown/research/ISSAC04/handout.pdf

[Col67] Collins, George E.: Subresultants and Reduced Polynomial Remainder Sequences. In: *J. ACM* 14 (1967), Januar, Nr. 1, S. 129. – ISSN 0004–5411

[Col75] Collins, George E.: Quantifier elimination for real closed fields by cylindrical algebraic decompostion. In: Brakhage, H. (Hrsg.): *Automata Theory and Formal Languages 2nd GI Conference Kaiserslautern, May 20–23, 1975*. Berlin, Heidelberg : Springer Berlin Heidelberg, 1975. – ISBN 978–3–540–37923–2, S. 134–183

[GKZ09] Gelfand, I.M. ; Kapranov, M. ; Zelevinsky, A.: *Discriminants, Resultants, and Multidimensional Determinants*. Birkhäuser Boston, 2009 (Modern Birkhäuser Classics). – 405 S. https://link.springer.com/book/10.1007%2F978-0-8176-4771-1. – ISBN 978–0–8176–4771–1

[Gri96] Griebl, Martin: *The Mechanical Parallelization of Loop Nests Containing* while *Loops*, University of Passau, Diss., 1996. http://www.uni-passau.de/~griebl/thesis.html. – also available as technical report MIP-9701

[Grö09] Grösslinger, Armin: *The Challenges of Non-linear Parameters and Variables in Automatic Loop Parallelisation*, University of Passau, Diss., 2009. https://opus4.kobv.de/opus4-uni-passau/files/116/Groesslinger_Armin.pdf

[HEW+15] Huang, Zongyan ; England, Matthew ; Wilson, David ; Davenport, James H. ; Paulson, Lawrence C.: A comparison of three heuristics to choose the variable ordering for CAD. In: *ACM Communications in Computer Algebra* 48 (2015), Nr. 3/4, 121 - 123. http://dx.doi.org/10.1145/2733693.2733706. – DOI 10.1145/2733693.2733706. – ISSN 1932–2240

[Hon90] Hong, H.: An Improvement of the Projection Operator in Cylindrical Algebraic Decomposition. In: *Proceedings of the International Symposium on Symbolic and Algebraic Computation*. New York, NY, USA : ACM, 1990 (ISSAC '90). – ISBN 0–201–54892–5, 261–264

[Len93] In: Lengauer, Christian: *Loop parallelization in the polytope model*. Berlin, Heidelberg : Springer Berlin Heidelberg, 1993. – ISBN 978–3–540–47968–0, 398–416

[McC85] McCallum, Scott: An improved projection operation for cylindrical algebraic decomposition. In: Caviness, Bob F. (Hrsg.): *EUROCAL '85*. Berlin, Heidelberg : Springer Berlin Heidelberg, 1985. – ISBN 978–3–540–39685–7, S. 277–278

[QRW00] Quilleré, Fabien ; Rajopadhye, Sanjay ; Wilde, Doran: Generation of Efficient Nested Loops from Polyhedra. In: *International Journal of Parallel Programming* 28 (2000), Oct, Nr. 5, 469–498. http://dx.doi.org/10.1023/A:1007554627716. – DOI 10.1023/A:1007554627716. – ISSN 1573–7640

[Sch86] Schrijver, Alexander: *Theory of Linear and Integer Programming*. New York, NY, USA : John Wiley & Sons, Inc., 1986. – ISBN 0–471–90854–1

[Str06] Strzeboński, Adam W.: Cylindrical Algebraic Decomposition using validated numerics. In: *Journal of Symbolic Computation* 41 (2006), Nr. 9, 1021 - 1038. http://dx.doi.org/https://doi.org/10.1016/j.jsc.2006.06.004. – DOI https://doi.org/10.1016/j.jsc.2006.06.004. – ISSN 0747–7171

[Syl40] Sylvester, James J.: XXIII. A method of determining by mere inspection the derivatives from two equations of any degree. In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 16 (1840), Nr. 101, 132-135. http://dx.doi.org/10.1080/14786444008649995. – DOI 10.1080/14786444008649995

[Tar48] Tarski, Alfred: *A Decision Method for a Elementary Algebra and Geometry*. Rand Corporation, 1948 (Project rand). https://books.google.at/books?id=gFptAAAAMAAJ

[TTSY00] Tanaka, Yoshizumi ; Taura, Kenjiro ; Sato, Mitsuhisa ; Yonezawa, Akinori: Performance Evaluation of OpenMP Applications with Nested Parallelism. In: *Selected Papers from the 5th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*. London, UK, UK : Springer-Verlag, 2000 (LCR '00). – ISBN 3–540–41185–2, 100–112

[VKÁ17] Viehmann, Tarik ; Kremer, Gereon ; Ábrahám, Erika: Comparing Different Projection Operators in the Cylindrical Algebraic Decomposition for SMT Solving. In: *SC^2@ISSAC*, 2017

[Xue00] In: Xue, Jingling: *Communication-Minimal Tiling*. Boston, MA : Springer US, 2000. – ISBN 978–1–4615–4337–4, 169–197

Statement of Authorship

I, Thomas Lang, hereby certify that this master thesis has been composed by myself and describes my own work unless otherwise stated. All references and verbatim extracts have been quoted and all sources of information have been specifically acknowledged. In addition, this thesis has not been accepted in any previous application for a degree.

Passau, 21st March 2018

____
(Thomas Lang)