Universität Passau
Fakultät für Mathematik und Informatik

Extending the Polyhedron Model to Inequality Systems with Non-linear Parameters using Quantifier Elimination

Diplomarbeit

Autor:
Armin Größlinger

Aufgabensteller:
Prof. Christian Lengauer, Ph.D.
Lehrstuhl für Programmierung
Universität Passau

Betreuer:
Dr. Martin Griebl

Passau, 23. September 2003

Abstract

The polyhedron model has proved to be a useful tool in studying methods for the automatic parallelization of loop nests. Most of the mathematical tools developed for the polyhedron model require the coefficients of variables to be constants. This restriction has turned out to be a severe limitation for several recent developments in the polyhedron model. We show how the polyhedron model can be generalized to allow non-linear parameters, i.e., parameters appearing in the coefficients of variables. The mathematical method we use for this generalization is quantifier elimination in the real numbers. We demonstrate how existing algorithms can be generalized by the use of decision methods and give examples of new algorithms which use quantifier elimination directly to solve common problems in the polyhedron model.

Acknowledgements

The work presented in this thesis has only been possible because of the support I got from the following people. First of all, I have to thank my tutor Dr. Martin Griebl for the many discussions about this thesis, his valuable suggestions, and for proofreading the text of this thesis. I am grateful to Prof. Christian Lengauer, Ph.D. for giving me the possibility to explore the topic treated here and the opportunity to write a diploma thesis about it. He read a draft of this thesis and improved my usage of the English language. My thanks also go to Dr. Andreas Dolzmann for his help with REDUCE/REDLOG and the discussions about quantifier elimination in general. I thank Peter Faber for reading an early draft of this thesis and for the discussions together with the other members of the *LooPo* team, Michael Classen, Tobias Langhammer, and Thomas Wondrak. Lorenz Lang helped me with some LaTeX-typesetting issues.

Contents

Chapter 1

Introduction

1.1 The Polyhedron Model

The field of automatic parallelization has been a research area for many years now. Different approaches to transform a sequential input program into a parallel target program have been developed. One of these is parallelization based on the *polyhedron model* [Len93]. In this approach the operations performed by the input program are described by polyhedra (subsets of the *n*-dimensional space of real numbers bounded by hyperplanes) or polytopes (bounded polyhedra) and a partial ordering on them, called the dependence relation. All the analysis and the transformations are performed in this mathematical model of the original program. The final target program is then generated from the resulting polyhedra. It is clear that the power and the limitations of this approach lie entirely in the expressivity of the polyhedron model and the transformations which can be performed therein.

```
for i := 0 to n do
  for j := 0 to i+2 do
    A(i,j) = A(i-1,j) + A(i,j-1);
  od
od
```

```
for t := 0 to 2n+2 do
  forall p := max(0,t-n) to min(t,⌊t/2⌋+1) do
    A(t-p,p) = A(t-p-1,p) + A(t-p,p-1);
  od
od
```

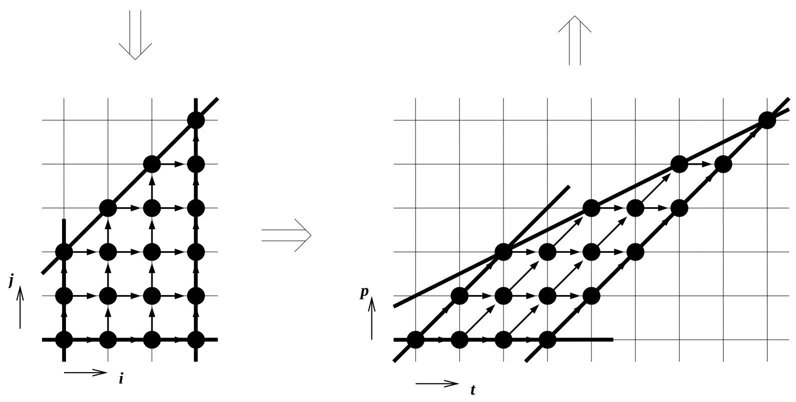


Figure 1.1: Loop transformations in the polyhedron model

Figure 1.1 shows a sketch of the complete process. From the program text a description of the source program in terms of a polyhedron is derived (left side). The polyhedron is then transformed (bottom) and a new program (the “target program”) is obtained from the resulting polyhedron (right side). This process happens at *compile time*, i.e., it is performed once when the target program is constructed and compiled.

Over the last decades powerful methods for the manipulation of polyhedra to achieve different goals (obtain parallelism, perform cache optimization, etc.) have been developed and used in compilers and meta-programming systems (e.g., *LooPo* [GL97]).

Most of the libraries and tools which have been developed for the polyhedron model (e.g., PIP [FCB02], Polylib [Wil93]) rest on the supposition that the variables of the equations and inequalities describing the polyhedra have constant coefficients. This is a reasonable choice for several reasons. First, the polyhedron model (and the older polytope model) were applied to describe programs which are nests of **for**-loops with affine bounds and uniform dependences [Lam74]. Even after the extensions from uniform to affine dependences and from the polytope model to the polyhedron model, many programs and the necessary transformations could still be described by systems with constant coefficients. Second, many algorithms depend on the signs of the coefficients, so non-constant coefficients would require the result to contain case distinctions (see Section 2.1.4).

However, for some problems the limitation to constant coefficients is too restrictive. As we will see in Section 1.2 and in greater detail in Section 5.2, the problem of tiling an index space of a program with a parametric tile size requires parameters to appear in the coefficients of some variables and can therefore not be solved with the existing techniques for systems with constant coefficients.

The main observations which motivated this thesis are:

(1) Generalizing existing algorithms of the polyhedron model to handle parametric coefficients requires the results of the algorithms to contain case distinctions with conditions depending on the non-linear parameters (Section 4.2).

(2) The case distinctions are in general very large and therefore must be simplified as much as possible.

(3) One method to simplify these case distinctions requires quantifier elimination (Chapter 3).

(4) Quantifier elimination can itself be used to compute some of the results often required when working in the polyhedron model (Section 4.3).

In this thesis we introduce some concepts from the theory of polyhedra, algebra, and logic (Chapter 2), introduce our main mathematical tool (quantifier elimination) in Chapter 3, elaborate on how we generalize the polyhedron model and some of its algorithms to non-linear parameters in Chapter 4, and show some applications of the generalized algorithms in Chapter 5.

1.2 Introductory Example with Non-linear Parameters

We illustrate the principal difficulty of introducing non-constant coefficients to the polyhedron model with a simple example. Suppose we are given the following program for $n \geq 0$:

for $i = 0$ **to** n
  $A[i] := f(i)$

This program assigns, for every $i \in \{0, \ldots, n\}$, a new value to $A[i]$ which is computed by the function call $f(i)$. We assume further that f does not have side effects, so the iterations of the **for**-loop are independent of each other. Therefore, it is possible to execute the iterations in an arbitrary order or even in parallel. What we want to do is execute the program on a processor grid with p processors, where $p \geq 1$.

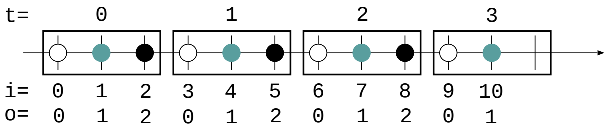


Figure 1.2: A tiled iteration domain

What we choose to do is partition the iteration domain $\{0,\dots,n\}$ of the program into parts (which are called *tiles*) of size p and, for every part, assign one iteration to each available processor. We are going to use tiles which span p adjacent points from the iteration domain. Figure 1.2 shows the iteration domain for $n = 10$ and the tiles we want to use for $p = 3$ (in boxes). The iterations which are to be performed on the same processor are drawn in the same color. The figure also shows that we assign each tile a number t. The offset of a point from the left side of the tile containing the point is called o. o is the number of the processor executing the respective iteration. In the depicted iteration domain, the offsets within a tile are 0, 1, 2, from left to right. Obviously, the offsets are, in general, in the range $0,\dots,p-1$.

Let us describe the tiling we desire through affine inequalities and one equation:

$$\begin{aligned} 0 \leq i &\leq n \\ 0 \leq o &\leq p-1 \\ i &= p \cdot t + o \end{aligned}$$

Note that the variables i and o have constant coefficients only, whereas in the equation $i = p \cdot t + o$ the variable t has the unknown coefficient p.

From this description we would like to extract two **for**-loops: the first loop enumerates the tiles and the second loop enumerates, for every tile enumerated by the first loop, the points of the iteration domain which lie inside the tile.[1]

Mathematically, this can be achieved by applying Fourier-Motzkin elimination (cf. Section 2.1.2) to the above system of inequalities and equations. We begin the elimination by substituting $o = i - p \cdot t$ (which is derived from the equation) into the inequalities. This yields

$$\begin{aligned} 0 &\leq i \leq n \\ 0 &\leq i - p \cdot t \leq p-1 \end{aligned}$$

The next step is to solve the inequalities for i:

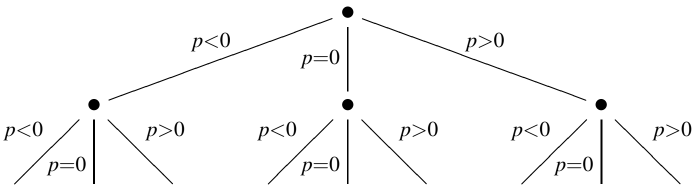
$$\begin{aligned} 0 &\leq i \\ p \cdot t &\leq i \\ i &\leq n \\ i &\leq p \cdot t + p - 1 \end{aligned} \tag{1.1}$$

[1]This loop structure has been chosen to keep the example simple. It does not yield an optimal parallel program.

Now we eliminate i from the system by comparing each lower bound against each upper bound:

$$
\begin{aligned}
0 &\leq n \\
0 &\leq p \cdot t + p - 1 \\
p \cdot t &\leq n \\
p \cdot t &\leq p \cdot t + p - 1
\end{aligned}
$$

Here we can do an optimization step. Since $n \geq 0$ and $p \geq 1$ is being assumed, the inequalities $0 \leq n$ and $p \cdot t \leq p \cdot t + p - 1$ are always true and can be removed. This step is not necessary for the correctness of the result, but it simplifies the following calculation. The next step is to solve the remaining inequalities ($0 \leq p \cdot t + p - 1$ and $p \cdot t \leq n$) for t, which is done by dividing the inequalities by p. This requires a case distinction on the sign of p since, when $p < 0$, the orientation of the inequality relation changes, and, when $p = 0$, dividing by p is not allowed (and the system does not restrict the values of t in any way). Since in both inequalities the coefficient of t is unknown, we have, in principle, to make two case distinctions:



Of course, 6 of the 9 possible cases are impossible because both case distinctions are on the sign of p. In addition, we have assumed $p \geq 1$, so the first case distinction on p is superfluous since $p \geq 1$ implies $p > 0$. The second case distinction on the sign p can be eliminated by the same reasoning or, alternatively, by the argument that $p > 0$ (the only possibility of the first case distinction) and this (trivially) implies that $p > 0$ when the sign of the "second p" is required. This reasoning is simple to carry out here, but it illustrates the general problem of avoiding unnecessary case distinctions.

Under the assumption of $p \geq 1$ the solved system takes the form

$$\frac{1}{p} - 1 \leq t \leq \frac{n}{p} \tag{1.2}$$

Systems (1.1) and (1.2) together give us all the bounds needed to construct a parallel program (see also Section 2.1.2):

for $t = \lceil \frac{1}{p} - 1 \rceil$ **to** $\lfloor \frac{n}{p} \rfloor$
  parfor $i = \max\{0, p \cdot t\}$ **to** $\min\{n, p \cdot t + p - 1\}$
    $A[i] := f(i)$

The outer **for**-loop enumerates the tiles t necessary to cover the iteration domain of the original program. The inner *parallel* **parfor**-loop enumerates every point from the iteration domain which lies in tile t and executes them in parallel. By construction, every tile contains p points, possibly except the last tile which may contain fewer points.

Although the example is relatively simple, some important observations can be made:

- The presence of a parameter in the coefficient of a variable (in this case the parameter p as coefficient of the variable t) introduces case distinctions in general (in our case on the sign of the parameter).
- The parameter n, which appears only in additive constants, does not cause case distinctions.
- To find out which of the case distinctions are really necessary, we have to analyze logical consequences: to find out that the cases $p < 0$ and $p = 0$ can never happen, we have to figure out that $p \geq 1$ implies $p > 0$ (in the real numbers).

Let us make the third point a bit more precise: what we had to find out exactly is, that in the real numbers the logical formula $\forall p\,(p \geq 1 \rightarrow p > 0)$ holds, i.e., for every real number p it is true that $p \geq 1$ implies $p > 0$. Note that it is necessary to use a quantifier to express this condition. The exact definition of logical formulas will be introduced in Section 2.2 and a methods to check (i.e., decide) quantified logical formulas in the reals is presented in Chapter 3.

Chapter 2

Mathematical Prerequisites

2.1 Polyhedra

2.1.1 Definitions

There are different mathematical ways to describe the concept of a polyhedron; the definition we use here is based on linear equations and inequalities:

Definition 2.1 A *linear equation* is an equation of the form

$$\sum_{i=1}^{n} c_i \cdot x_i + d = 0$$

where $x_1, \ldots, x_n$ are the unknowns and $c_1, \ldots, c_n, d \in \mathbb{R}$. An inequality of the form

$$\sum_{i=1}^{n} c_i \cdot x_i + d \geq 0$$

is called a *linear inequality*. $c_1, \ldots, c_n$ are called *coefficients*, and d is called the *additive constant* or *additive term*.

Linear equations can be expressed by two linear inequalities, since $\sum_{i=1}^{n} c_i x_i + d = 0$ is equivalent to $\sum_{i=1}^{n} c_i x_i + d \geq 0 \wedge \sum_{i=1}^{n} c_i x_i + d \leq 0$. It is desirable to use equations whenever possible, since equations can be treated much more efficiently by many algorithms than a pair of complementary inequalities (see, for example, Section 4.2.1).

We use also a vectorial notation for linear equations and inequalities: with $\bar{x} = (x_1, \ldots, x_n)$ and $\bar{c} = (c_1, \ldots, c_n)$, they can be written as follows:

$$\bar{c} \cdot \bar{x} + d = 0$$
$$\bar{c} \cdot \bar{x} + d \geq 0$$

or even (using homogeneous coordinates) as follows:

$$\begin{pmatrix} \bar{c} & d \end{pmatrix} \cdot \begin{pmatrix} \bar{x} \\ 1 \end{pmatrix} \geq 0$$

The algebraic concepts of a linear equation and inequality have direct geometric correspondents:

Definition 2.2 Let $\bar{a} \in \mathbb{R}^n \setminus \{0\}$ and $b \in \mathbb{R}$. Then

$$A(\bar{a}, b) = \{\bar{x} \in \mathbb{R}^n \mid \bar{a} \cdot \bar{x} + b = 0\}$$

is called a *hyperplane*.

A fundamental result of the theory of linear algebra states that the hyperplanes of $\mathbb{R}^n$ are exactly the affine subspaces of $\mathbb{R}^n$ with dimensionality $n-1$ (see any linear algebra book).

Definition 2.3 Let $\bar{a} \in \mathbb{R}^n \setminus \{0\}$ and $b \in \mathbb{R}$. The set

$$H(\bar{a}, b) = \{\bar{x} \in \mathbb{R}^n \mid \bar{a} \cdot \bar{x} + b \geq 0\}$$

is called a *halfspace* of $\mathbb{R}^n$.

Please note that $A(\bar{a}, b) = A(-\bar{a}, -b)$, for any $\bar{a}$ and b, but $H(\bar{a}, b) \neq H(-\bar{a}, -b)$. Furthermore:

$$\begin{aligned} H(\bar{a}, b) \cap H(-\bar{a}, -b) &= A(\bar{a}, b) \\ H(\bar{a}, b) \cup H(-\bar{a}, -b) &= \mathbb{R}^n \end{aligned}$$

From the correspondence between linear equations and hyperplanes it is immediately obvious that the same correspondence holds between linear inequalities and halfspaces.

Definition 2.4 A *polyhedron* P in n-dimensional real space is a subset of $\mathbb{R}^n$ which is an intersection of finitely many halfspaces, i.e., for $m \in \mathbb{N}$, $\bar{a}_1, \ldots, \bar{a}_m \in \mathbb{R}^n \setminus \{0\}$, and $b_1, \ldots, b_m \in \mathbb{R}$:

$$P = \bigcap_{i=1}^{m} H(\bar{a}_i, b_i) = \{\bar{x} \in \mathbb{R}^n \mid \bigwedge_{i=1}^{m} \bar{a}_i \cdot \bar{x}_i + b_i \geq 0\}$$

An alternative form of describing the polyhedron is

$$P = \{\bar{x} \in \mathbb{R}^n \mid M \cdot \bar{x} + \bar{b} \geq 0\}$$

where

$$M = \begin{pmatrix} \bar{a_1} \\ \vdots \\ \bar{a_m} \end{pmatrix}, \qquad \bar{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$$

Definition 2.5 A bounded polyhedron is called a *polytope*.

2.1.2 Fourier-Motzkin Elimination

One of the fundamental algorithms which can be applied to polyhedra is *Fourier-Motzkin* elimination ([Ban93], Section 3.8; [Sch94], Section 12.2). Given a finite set S of inequalities with variables $x_1, \ldots, x_n$ and *linear* parameters $p_1, \ldots, p_m$ as input, this algorithm calculates the following two pieces of information:

- a set F of inequalities only in the parameters such that $\bigwedge F$ (i.e., the conjunction of the inequalities in F) is true for given values of the parameters if and only if the polyhedron described by $\bigwedge S$ is non-empty, and

- inequality systems L_i and U_i representing lower and upper bounds of x_i (for $1 \leq i \leq n$) such that

  - L_i contains only lower bounds of x_i, i.e., inequalities of the form $x_i \geq t$ for some expression t (which does not contain x_i),
  - U_i contains only upper bounds of x_i, i.e., inequalities of the form $x_i \leq t$ for some expression t (wherein x_i does not occur),
  - $\bigwedge S$ is equivalent to $\bigwedge_{i=1}^{n}(\bigwedge L_i \wedge \bigwedge U_i)$,
  - no variable $x_{i+1},\ldots,x_n$ appears in U_i or L_i, for $1 \leq i < n$.

This means that L_1 and U_1 contain inequalities representing lower and upper bounds for x_1 which depend only on the parameters. L_2 and U_2 list lower and upper bounds for x_2 in dependence of x_1 and the parameters, and so forth.

Given an affine expression $t := \sum_{i=1}^{n} c_i x_i + \sum_{i=1}^{m} d_i p_i + e$ with the variables $x_1,\ldots,x_n$, parameters $p_1,\ldots,p_m$, and additive constant e, we write $\mathrm{coeff}_{x_i}(t)$ to denote c_i, the coefficient of x_i.

The algorithm works as follows:

$$
\begin{aligned}
S_n &:= S \\
L_i &:= \{x_i \geq x_i - \frac{t}{\mathrm{coeff}_{x_i}(t)} \mid (t \geq 0) \in S_i, \mathrm{coeff}_{x_i}(t) > 0\} && \text{for } i = n,\ldots,1 \\
U_i &:= \{x_i \leq x_i - \frac{t}{\mathrm{coeff}_{x_i}(t)} \mid (t \geq 0) \in S_i, \mathrm{coeff}_{x_i}(t) < 0\} && \text{for } i = n,\ldots,1 \\
S_{i-1} &:= \{t \geq 0 \mid (t \geq 0) \in S_i, \mathrm{coeff}_{x_i}(t) = 0\} \cup && \text{for } i = n,\ldots,1 \\
&\quad\ \{t' - t \geq 0 \mid (x_i \geq t) \in L_i, (x_i \leq t') \in U_i\} \\
F &:= S_0
\end{aligned}
$$

Note that x_i cancels in the expressions $x_i - \frac{t}{\mathrm{coeff}_{x_i}(t)}$, so L_i and U_i really contain inequalities with the specified properties.

As stated above, the condition $\bigwedge F$ states (in dependence of the parameters) whether $\bigwedge S$ has a solution or not. It is important to note that any solution is in the *reals*. If condition $\bigwedge F$ is satisfied by the parameters, it is still possible that $\bigwedge S$ has no integral solution. Algorithms deciding the existence of integral solutions are beyond the scope of this thesis. We just show a way to enumerate the integral points of a polyhedron in the next section.

Unfortunately, Fourier-Motzkin elimination can suffer from severe performance problems in practice. This is due to the fact that—in the worst case—the number of inequalities is squared in every elimination step: if a given system has b bounds, the worst case is when half of the inequalities are lower bounds and the other half are upper bounds. The number of inequalities in the following step is then $\frac{b}{2} \cdot \frac{b}{2} = \frac{b^2}{4}$. Therefore, Fourier-Motzkin elimination is at least doubly exponential in the number of variables in the worst case. We discuss how to constrain this growth of inequalities in Section 4.2.1.

2.1.3 Polyhedra and Integers

In many applications one is interested in the integral points inside a polyhedron. Unfortunately, there is no *simple* technique to decide whether a polyhedron contains at least one integral point (in dependence of the parameters), so we do not discuss this question here. Instead, we present a method to enumerate all integral points of a polyhedron. This means that, in the polyhedron model, we are working in the reals most of the time and only come back to the integers in the final step when code to enumerate the integral points of the derived polyhedra is generated.

Let S again be a set of inequalities in the variables $x_1,\dots,x_n$. Fourier-Motzkin elimination computes the sets of lower and upper bounds

$$L_i = \{x_i \geq l_{i,1}(x_1,\dots,x_{i-1}),\dots,x_i \geq l_{i,k_i}(x_1,\dots,x_{i-1})\}$$
$$U_i = \{x_i \leq u_{i,1}(x_1,\dots,x_{i-1}),\dots,x_i \leq u_{i,o_i}(x_1,\dots,x_{i-1})\}$$

for every variable $x_1,\dots,x_n$ such that $\bigwedge S$ is equivalent to

$$\begin{aligned}
l_{1,1},\dots,l_{1,k_1} &\leq x_1 \leq u_{1,1},\dots,u_{1,o_1}\\
l_{2,1}(x_1),\dots,l_{2,k_2}(x_1) &\leq x_2 \leq u_{2,1}(x_1),\dots,u_{2,o_2}(x_1)\\
&\vdots\\
l_{n,1}(x_1,\dots,x_{n-1}),\dots,l_{n,k_n}(x_1,\dots,x_{n-1}) &\leq x_n \leq u_{n,1}(x_1,\dots,x_{n-1}),\dots,u_{n,o_n}(x_1,\dots,x_{n-1})
\end{aligned}$$

Of course, the parameters $p_1,\dots,p_m$ can appear in every $l_{i,j}$ and $u_{i,j}$, but the dependence of the bounds on the parameters is left out in the notation used here for readability purposes.

We replace every lower bound l by $\lceil l \rceil$ and every upper bound u by $\lfloor u \rfloor$ and enumerate only integral values for every variable between the (integral) lower and upper bounds. This is usually done by constructing a nest of **for**-loops:

for $x_1 = \max\{\lceil l_{1,1}\rceil,\dots,\lceil l_{1,k_1}\rceil\}$ **to** $\min\{\lfloor u_{1,1}\rfloor,\dots,\lfloor u_{1,o_1}\rfloor\}$
  for $x_2 = \max\{\lceil l_{2,1}(x_1)\rceil,\dots,\lceil l_{2,k_2}(x_1)\rceil\}$ **to** $\min\{\lfloor u_{2,1}(x_1)\rfloor,\dots,\lfloor u_{2,o_2}(x_1)\rfloor\}$
    $\ddots$
      for $x_n = \max\{\lceil l_{1,n}(x_1,\dots,x_{n-1})\rceil,\dots,\lceil l_{n,k_n}(x_1,\dots,x_{n-1})\rceil\}$ **to**
        $\min\{\lfloor u_{n,1}(x_1,\dots,x_{n-1})\rfloor,\dots,\lfloor u_{n,o_n}(x_1,\dots,x_{n-1})\rfloor\}$
        loop body depending on $(x_1,\dots,x_n)$

This loop nest enumerates all points $(x_1,\dots,x_n) \in \mathbb{Z}^n$ which satisfy $\bigwedge S$ (for given values of the parameters $p_1,\dots,p_m$). If the enumerated polyhedron has no integral solutions (but some real solutions) at least one of the **for**-loops will be empty (in the sense that its lower bound is greater than its upper bound). This means that some superfluous work is done by the outer **for**-loops, but the enumeration is correct and complete in every case.

This method to enumerate the integral points of a polyhedron can be used together with the generalized Fourier-Motzkin algorithm for inequality systems with non-linear parameters, which we present in Section 4.2.1.

For our algorithm to compute disjoint unions of polyhedra (given in Section 4.3.3), we need a trivial correspondence between strict and weak inequalities with respect to integral solutions:

Lemma 2.6 *Let $\sum_{i=1}^{n} c_i \cdot x_i + d$ be an affine expression in the variables $x_1, \ldots, x_n$ and coefficients $c_1, \ldots, c_n, d \in \mathbb{Z}$. Then, for every $(x_1, \ldots, x_n) \in \mathbb{Z}$,*

$$\sum_{i=1}^{n} c_i \cdot x_i + d > 0 \quad \textit{if and only if} \quad \sum_{i=1}^{n} c_i \cdot x_i + d \geq 1$$

and

$$\sum_{i=1}^{n} c_i \cdot x_i + d \neq 0 \quad \textit{if and only if} \quad \left(\sum_{i=1}^{n} c_i \cdot x_i + d \geq 1 \vee \sum_{i=1}^{n} c_i \cdot x_i + d \leq -1 \right)$$

This lemma allows us to replace strict inequalities by weak inequalities (or a disjunction of weak inequalities if we replace an inequality with the relation $\neq$), provided that the coefficients and the additive constant are integral numbers and we are only interested in integral solutions of the inequality. That the coefficients are integral does not mean that they have to be constant numbers from $\mathbb{Z}$; for example, they can also be polynomials in the parameters if the parameters are assumed to be integral, too.

2.1.4 Polyhedra with Parameters

The definition of polyhedra given above states that the coefficients of the variables and the additive constant be real numbers. From the mathematical point of view, there is little difference between using fixed numbers as coefficients, as in $2 \cdot x - 4 \geq 0$, and specifying the coefficients through parameters, as in $p \cdot x - q \geq 0$. Unfortunately, from the computational point of view, there is a big difference between these two inequalities. This becomes clear when one tries to solve either inequality for the only variable x.

In the case of $2 \cdot x - 4 \geq 0$, the coefficient of x is a fixed number, namely 2, and it can be determined statically (at compile time) that $2 > 0$. Therefore, solving the inequality for x yields $x \geq \frac{4}{2}$.

On the other hand, the coefficient of x in $p \cdot x - q \geq 0$ is the parameter p, whose sign cannot be determined (without further assumptions), so to solve for x in the general case requires a case distinction to be made:

- Case 1: $p > 0$. We can divide by p and the orientation of the relation symbol stays the same: $x \geq \frac{q}{p}$.
- Case 2: $p < 0$. We can divide by p, but have to change the orientation of the relation symbol: $x \leq \frac{q}{p}$.
- Case 3: $p = 0$. The inequality is in this case just the condition $q \geq 0$, so we cannot solve for x here.

Looking more closely at the problems introduced by parameters, it becomes clear that different levels of parametrization can be distinguished:

(1) The simplest extension of practical interest is to allow parameters to appear *linearly* in the additive term,

$$\sum_{i=1}^{n} c_i x_i + \sum_{j=1}^{m} d_i p_i + e \geq 0 \tag{2.1}$$

where $x_1,\dots,x_n$ are again the variables, $p_1,\dots,p_m$ are the parameters (called *weak parameters* as they appear only linearly), and the coefficients $c_1,\dots,c_n,d_1,\dots,d_m \in \mathbb{R}$ and the additive constant $e \in \mathbb{R}$ are fixed real numbers. Handling this case is relatively easy, and it has been done for years in the domain of automatic loop parallelization. The reason that linear parameters are no big problem is that one can interpret Inequality (2.1) as being non-parametric and having the $n+m$ variables $x_1,\dots,x_n,p_1,\dots,p_m$. Then it is possible to apply all computation techniques for the non-parametric polyhedron model.

(2) A step further in the direction of full parametrization would be to allow parameters to appear non-linearly in the additive term, i.e.,

$$\sum_{i=1}^{n} c_i x_i + f \geq 0 \tag{2.2}$$

where f is an arbitrary polynomial in the parameters $p_1,\dots,p_m$. But this case can be reduced to the case with linear parameters only by introducing new linear parameters for every polynomial appearing in the input inequalities. For example,

$$2x + p^2 + 3 \geq 0$$
$$3x - p^3 + 1 \geq 0$$

can be rewritten as

$$2x + q_1 \geq 0$$
$$3x + q_2 \geq 0$$

with $q_1 := p^2 + 3$ and $q_2 := -p^3 + 1$ and the transformed system can be handled as in case (1). After the computations are performed, the linear parameters q_1 and q_2 are resubstituted with the original expressions in p.

(3) The most general and most desirable form of parametrization is to introduce *strong parameters*, which can appear in arbitrary powers and products in the coefficients of a variable or in the additive term. Then the inequalities take again the form

$$\sum_{i=1}^{n} c_i x_i + d \geq 0 \tag{2.3}$$

but now $c_1,\dots,c_n,d$ are polynomials in $p_1,\dots,p_m$ which are the parameters of the polyhedron.

The big difference between (1), (2) on the one hand and (3) on the other is the effect the parameters can have on the hyperplanes and halfspaces of one inequality (or the hyperplane of one equation, respectively).

Parameters appearing in the additive term can only translate hyperplanes (and therefore halfspaces) in space, but the direction does not change. This is clear from the theory of linear algebra, since the coefficients $c_1,\dots,c_n$ describe a normal of the hyperplane (which is constant) and the additive constant determines the translation of the hyperplane.

In contrast, when parameters appear in the coefficients of variables, the normal depends on parameters and its orientation (and accordingly the direction of the hyperplane) changes

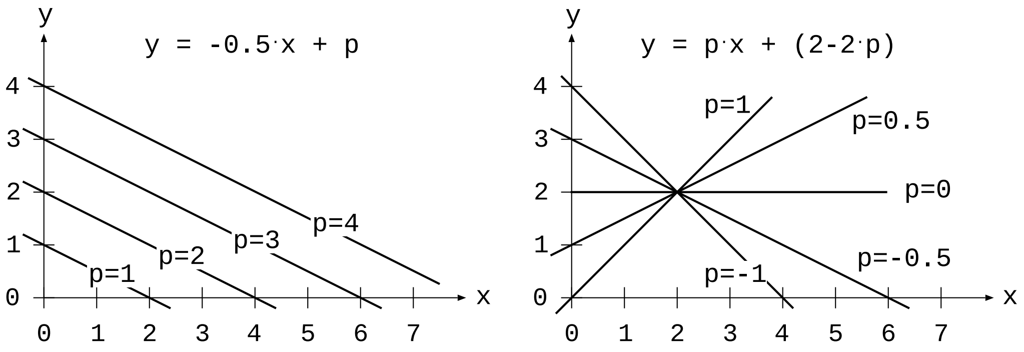


Figure 2.1: Hyperplanes defined by equations with a linear and a non-linear parameter

as the parameters change. This is illustrated in Figure 2.1. The left diagram shows the hyperplanes $y = \frac{1}{2} \cdot x + p$ for some p, and obviously they are all parallel. The right diagram shows that the hyperplanes defined by $y = p \cdot x + (2 - 2 \cdot p)$ are not parallel and their directions change with p, since p appears as coefficient of the variable x.

The example $p \cdot x - q \geq 0$ at the beginning of this section shows that the orientation of the solution space (which is a halfspace) depends on the sign of p. This example also shows that it is not sufficient to allow polynomials in the parameters as coefficients of variables. To solve the inequality $p \cdot x - q \geq 0$ for x we have to use the fraction $\frac{q}{p}$.

Another complication is that we cannot assume that the parameters do not "mix", i.e., if different parameters appear in the coefficients of different variables in the input, they can—after some transformations—appear together in the same coefficient. Let us take

$$\begin{aligned} p \cdot x + y &\geq 0 \\ x - q \cdot y &\leq 0 \end{aligned}$$

as input of an elimination step which is, e.g., found in the Fourier-Motzkin algorithm (see Section 2.1.2). To eliminate x, we first solve the system for x (assuming $p > 0$):

$$\begin{aligned} -\frac{1}{p} \cdot y \leq x& \\ x &\leq q \cdot y \end{aligned}$$

and then we eliminate x by comparing the lower bound of x against the upper bound:

$$-\frac{1}{p} \cdot y \leq q \cdot y$$

After merging we obtain the result:

$$\left(q + \frac{1}{p}\right) \cdot y \geq 0$$

In the input, the parameter p appears only as coefficient of x and q only appears as coefficient of y, but after the elimination of x both parameters appear together in the coefficient of y.

These two observations show that, if we introduce non-linear parameters, the right choice for the domain of the coefficients is the quotient field $\mathbb{Q}(p_1, \ldots, p_n)$, which consists of fractions of polynomials from the polynomial ring $\mathbb{Q}[p_1, \ldots, p_n]$. Formal definitions of $\mathbb{Q}[p_1, \ldots, p_n]$ and $\mathbb{Q}(p_1, \ldots, p_n)$ are given in Section 2.3.3.

2.2 Logic

As we have seen in the introductory example in Section 1.2, concepts from the theory of logic are needed in dealing with our problem, like implication and (first-order) quantification. This section introduces some notation common in logic, which will be the base of our formal approach to dealing with logical formulas in subsequent sections.

We need ordinary first-order logic. "Ordinary" is used to distinguish this logic from its variants like intuitionistic logic. Ordinary first-order logic is the most commonly used logic in mathematics and computer science, so the reader should be familiar with it. "First-order" means that quantification is possible only for individuals, i.e., for a single value, not for sets.

To deal formally with logical formulas, we must first introduce the notion of a (logical) *language*. The language determines which expressions ("terms") and formulas can be legally formed in the logic. After that, a semantics has to be assigned to the formulas.

The base for the definition of the language is a *signature*. A signature is a common concept in computer science and it plays the same role in logic. Our signatures are relatively simple: all operands are of the same "type," and no higher-order functions are allowed.

Definition 2.7 A *signature* Σ is a triple $(\mathcal{F}, \mathcal{R}, ar)$ consisting of the set of *function symbols* $\mathcal{F}$, the set of relation symbols $\mathcal{R}$, and an *arity assignment* $ar : \mathcal{F} \cup \mathcal{R} \to \mathbb{N}$.

The meaning of the signature is simple: we select some symbols we want to use to denote functions, e.g., $\mathcal{F} = \{+, -, \cdot, 0, 1\}$, and relations, e.g., $\mathcal{R} = \{<\}$, and define the arities of the function and relation symbols, e.g., $ar(+) = ar(\cdot) = ar(<) = 2$, $ar(-) = 1$, $ar(0) = ar(1) = 1$. This example gives us a signature with the two binary operations "$+$" (addition), "$\cdot$" (multiplication), the unary function "$-$" (unary negation), two nullary function symbols (also called constants) "0", "1", and a binary relation "$<$" (less than). This signature $\Sigma_{ord} = (\mathcal{F}, \mathcal{R}, ar)$ is often called the *signature of ordered rings*. We come back to this signature in subsequent sections.

It is important to note here that the signature does not assign any meaning (i.e., semantics) to the function and relation symbols. The concrete meaning of the symbols is determined by an interpretation (see below).

Terms and formulas will have to contain variables to be useful, so we need an infinite set $\mathcal{V}$ containing all variables. The alphabet of the logical language is:

$$L = \mathcal{V} \cup \mathcal{F} \cup \mathcal{R} \cup \{(,),,,=\}$$

Now we can define the set of *terms* for the signature Σ.

Definition 2.8 $Tm(\mathcal{V}, \Sigma)$, the set of *terms* over $\mathcal{V}$ and Σ, is inductively defined by:

(1) $\mathcal{V} \subseteq Tm(\mathcal{V}, \Sigma)$

(2) if $f \in \mathcal{F}$ and $ar(f) = 0$ then $f \in Tm(\mathcal{V}, \Sigma)$

(3) if $f \in \mathcal{F}$, $ar(f) = n > 0$ and $t_1, \ldots, t_n \in Tm(\mathcal{V}, \Sigma)$ then $f(t_1, \ldots, t_n) \in Tm(\mathcal{V}, \Sigma)$

This definition states that every variable is a term, every nullary function symbol is a term, and every function symbol applied to an appropriate number of other terms is again a term. To continue the above example, legal terms for the signature Σ_{ord} are, e.g.,

$$x \qquad +(x, 1) \qquad *(y, +(-(x), 0))$$

with $x, y \in \mathcal{V}$.

The prefix notation, e.g., $+(x,1)$, is tedious, so the more common infix notation $x+1$ is used. We also apply the usual precedence rules to save parenthesis, e.g., we write $y*(-x+0)$ to denote the formal term $*(y,+(-(x),0))$. In addition, $x-y$ will be used as an abbreviation for $x+(-y)$.

Atomic formulas come in two variants: equations and relations.

Definition 2.9 $At(\mathcal{V},\Sigma)$, the set of *atomic formulas*, consists of:

(1) equations $t_1 = t_2$ where $t_1, t_2 \in Tm(\mathcal{V},\Sigma)$,

(2) truth constants r where $r \in \mathcal{R}$, $ar(r) = 0$, and

(2) predicates $r(t_1,\ldots,t_n)$ where $r \in \mathcal{R}$, $n = ar(r) > 0$.

As with the terms, we usually write relation symbols in infix notation. Using Σ_{ord}, for example, we write $x < y$ instead of $<(x,y)$.

Definition 2.10 $Fm(\mathcal{V},\Sigma)$, the set of *first-order formulas*, is inductively defined by:

(1) $At(\mathcal{V},\Sigma) \subseteq Fm(\mathcal{V},\Sigma)$

(2) if $\varphi \in Fm(\mathcal{V},\Sigma)$ then $(\neg\varphi) \in Fm(\mathcal{V},\Sigma)$

(3) if $\varphi, \psi \in Fm(\mathcal{V},\Sigma)$ and $\rho \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$ then $(\varphi\,\rho\,\psi) \in Fm(\mathcal{V},\Sigma)$

(4) if $x \in \mathcal{V}$, $\varphi \in Fm(\mathcal{V},\Sigma)$ then $(\exists x \varphi), (\forall x \varphi) \in Fm(\mathcal{V},\Sigma)$

$Qf(\mathcal{V},\Sigma)$, the set of *quantifier-free formulas*, is the subset of $Fm(\mathcal{V},\Sigma)$ which is obtained when rule (4) is not used in the formula construction, i.e., the formulas in $Qf(\mathcal{V},\Sigma)$ do not contain $\exists x$ or $\forall x$ constructs.

Again, this definition imposes strict requirements where to put parenthesis, but we apply the usual rules to save parenthesis: in the chain $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$ the symbol $\neg$ shall have the highest, and the symbol $\leftrightarrow$ shall have the lowest precedence.

A formula for a given signature is just syntax; a semantics is only associated with a formula when we choose a structure in which the formula is to be interpreted. This is analogous to the signature/structure concept in computer science.

A structure for a given signature Σ defines the domain (or universe) in which the formula is interpreted and defines mappings from the function and relation symbols of Σ to functions and relations over the domain of the structure.

Definition 2.11 A Σ-*structure* $\underline{A}$ is a triple $(A, i_{\mathcal{F}}, i_{\mathcal{R}})$ where

(1) A is a non-empty set (i.e., $A \neq \emptyset$),

(2) $i_{\mathcal{F}} : \mathcal{F} \rightarrow \bigcup_{n\in\mathbb{N}} A^{(A^n)}$ with $ar(f) = n$ implies $i_{\mathcal{F}}(f) \in A^{(A^n)}$,

(3) $i_{\mathcal{R}} : \mathcal{R} \rightarrow \bigcup_{n\in\mathbb{N}} 2^{(A^n)}$ with $ar(r) = n$ implies $i_{\mathcal{R}}(r) \in 2^{(A^n)}$

As a short-hand notation we define $f^{\underline{A}} := i_{\mathcal{F}}(f)$ for $f \in \mathcal{F}$ and $r^{\underline{A}} := i_{\mathcal{R}}(r)$ for $r \in \mathcal{R}$.

What a structure $\underline{A}$ really does (besides defining a universe A), is that it assigns to every n-ary function symbol f a function $f^{\underline{A}} : A^n \to A$ and to every n-ary relation symbol r a relation $r^{\underline{A}} \subseteq A^n$.

The interpretation of function symbols defined by a structure $\underline{A}$ extends naturally to terms if we also choose an *environment*, i.e., a mapping $h : \mathcal{V} \to A$ which assigns to every variable a value from the domain A of the structure $\underline{A}$.

Definition 2.12 Given a term $t \in Tm(\mathcal{V}, \Sigma)$, its interpretation $t_h^{\underline{A}}$ in a structure $\underline{A} = (A, i_{\mathcal{F}}, i_{\mathcal{R}})$ in the environment $h : \mathcal{V} \to A$ is defined as follows:

(1) $t_h^{\underline{A}} = h(t)$ if $t \in \mathcal{V}$

(2) $t_h^{\underline{A}} = f^{\underline{A}}$ if $t \in \mathcal{F}$, $ar(f) = 0$

(3) $t_h^{\underline{A}} = f^{\underline{A}}\big((t_1)_h^{\underline{A}}, \ldots, (t_n)_h^{\underline{A}}\big)$ if $t = f(t_1, \ldots, t_n)$, $ar(f) = n > 0$

The interpretation of a term (in a given environment) is a value from the domain of the structure. The interpretation of a formula is a truth value, i.e., the value "true" (denoted by $\top$) if the formula holds under the given environment, and "false" (denoted by $\bot$) otherwise. Using the interpretation of terms, the interpretation of quantifier-free formulas is straight forward:

Definition 2.13 Given a quantifier-free formula $\varphi \in Qf(\mathcal{V}, \Sigma)$, its interpretation $\varphi_h^{\underline{A}}$ in a structure $\underline{A} = (A, i_{\mathcal{F}}, i_{\mathcal{R}})$ in an environment $h : \mathcal{V} \to A$ is defined as follows by induction on the formula structure:

(1) $(t = t')_h^{\underline{A}} = \begin{cases} \top & \text{if } t_h^{\underline{A}} = t'^{\underline{A}}_h \\ \bot & \text{otherwise} \end{cases}$

(2) if $\varphi = r$, $r \in \mathcal{R}$ then $\varphi_h^{\underline{A}} = r^{\underline{A}}$

(3) if $\varphi = r(t_1, \ldots, t_n)$ then $\varphi_h^{\underline{A}} = \begin{cases} \top & \text{if } \big((t_1)_h^{\underline{A}}, \ldots, (t_n)_h^{\underline{A}}\big) \in r^{\underline{A}} \\ \bot & \text{otherwise} \end{cases}$

(4) $(\neg\varphi)_h^{\underline{A}} = \begin{cases} \top & \text{if } \varphi_h^{\underline{A}} = \bot \\ \bot & \text{otherwise} \end{cases}$

(5) $(\varphi \wedge \psi)_h^{\underline{A}} = \begin{cases} \top & \text{if } \varphi_h^{\underline{A}} = \top \text{ and } \psi_h^{\underline{A}} = \top \\ \bot & \text{otherwise} \end{cases}$

(6) $(\varphi \vee \psi)_h^{\underline{A}} = \begin{cases} \top & \text{if } \varphi_h^{\underline{A}} = \top \text{ or } \psi_h^{\underline{A}} = \top \\ \bot & \text{otherwise} \end{cases}$

(7) $(\varphi \to \psi)_h^{\underline{A}} = \begin{cases} \top & \text{if } \varphi_h^{\underline{A}} = \bot \text{ or } \psi_h^{\underline{A}} = \top \\ \bot & \text{otherwise} \end{cases}$

(8) $(\varphi \leftrightarrow \psi)_h^{\underline{A}} = \begin{cases} \top & \text{if } \varphi_h^{\underline{A}} = \psi_h^{\underline{A}} \\ \bot & \text{otherwise} \end{cases}$

The semantics of formulas with quantifiers is defined by modifying the environment. For $h, h' : \mathcal{V} \to A$ the notation $h' =_x h$ shall mean that, for all $y \in \mathcal{V} \setminus \{x\}$, $h'(y) = h(y)$, i.e., h and h' are the same on all variables, possibly except x.

(9) $(\exists x \varphi)_h^A = \begin{cases} \top & \text{if there is an environment } h' \text{ with } h' =_x h \text{ and } \varphi_{h'}^A = \top \\ \bot & \text{otherwise} \end{cases}$

(10) $(\forall x \varphi)_h^A = \begin{cases} \top & \text{if } \varphi_{h'}^A = \top \text{ for every environment } h' \text{ with } h' =_x h \\ \bot & \text{otherwise} \end{cases}$

$\varphi_h^A = \top$ is also written as $\underline{A} \models_h \varphi$. If $\varphi_h^A = \top$ for *every* environment h, we write $\underline{A} \models \varphi$.

We must deal with the variables of a term, or the variables appearing freely (unquantified) in a formula.

Definition 2.14 $var(t)$, the *set of variables appearing in a term t*, is defined by:

(1) $var(x) = \{x\}$ if $x \in \mathcal{V}$

(2) $var(f) = \varnothing$ if $f \in \mathcal{F}$ and $ar(f) = 0$

(3) $var\big(f(t_1, \ldots, t_n)\big) = \bigcup_{i=1}^{n} var(t_i)$ if $f \in \mathcal{F}$ and $ar(f) = n > 0$

The *set of free variables frvar*(φ) of a formula $\varphi \in Fm(\mathcal{V}, \Sigma)$ is defined by:

(1) $frvar(t_1 = t_2) = var(t_1) \cup var(t_2)$

(2) $frvar(r) = \varnothing$ if $r \in \mathcal{R}$ and $ar(r) = 0$

(3) $frvar\big(r(t_1, \ldots, t_n)\big) = \bigcup_{i=1}^{n} var(t_i)$ if $r \in \mathcal{R}$ and $ar(r) = n > 0$

(4) $frvar(\neg\varphi) = frvar(\varphi)$

(5) $frvar(\varphi \rho \psi) = frvar(\varphi) \cup frvar(\psi)$ for $\rho \in \{\wedge, \vee, \to, \leftrightarrow\}$

(6) $frvar(Qx\varphi) = frvar(\varphi) \setminus \{x\}$ for $Q \in \{\forall, \exists\}$ and $x \in \mathcal{V}$

It is sometimes easier to use an alternative notation of "extended terms" and "extended formulas" instead of environments when talking about the values of formulas.

Definition 2.15 Given a term $t \in Tm(\mathcal{V}, \Sigma)$ and pairwise different $x_1, \ldots, x_n$ with $var(t) \subseteq \{x_1, \ldots, x_n\}$, $t(x_1, \ldots, x_n)$ is called an *extended term*, and $(x_1, \ldots, x_n)$ is called an *extension*.
Given a formula $\varphi \in Fm(\mathcal{V}, \Sigma)$ and pairwise different $x_1, \ldots, x_n \in \mathcal{V}$ with $frvar(\varphi) \subseteq \{x_1, \ldots, x_n\}$, $\varphi(x_1, \ldots, x_n)$ is called an *extended formula*, and $(x_1, \ldots, x_n)$ is called an *extension*.

When we select an extension of a term or a formula, it is not necessary to specify a complete environment to determine the value of the term or formula, but it is sufficient to specify values for the variables of the extension.

Definition 2.16 Given a term $t \in Tm(\mathcal{V},\Sigma)$ or a formula $\varphi \in Fm(\mathcal{V},\Sigma)$ and a fixed extension $(x_1,\ldots,x_n)$, a structure $\underline{A}$ with universe A and elements $a_1,\ldots,a_n$, we define

$$t^{\underline{A}}(a_1,\ldots,a_n) := t_h^{\underline{A}}$$
$$\varphi^{\underline{A}}(a_1,\ldots,a_n) := \varphi_h^{\underline{A}}$$

for any environment $h : \mathcal{V} \to A$ with $h(x_1) = a_1,\ldots,h(x_n) = a_n$.

We note here that:

- The values of $t^{\underline{A}}(a_1,\ldots,a_n)$ and $\varphi^{\underline{A}}(a_1,\ldots,a_n)$ depend on the choice of the extension $x_1,\ldots,x_n$. Therefore, we say that we fix an extension before talking about $t^{\underline{A}}(a_1,\ldots,a_n)$ or $\varphi^{\underline{A}}(a_1,\ldots,a_n)$.
- The definition of an extension (Definition 2.15) ensures that $t^{\underline{A}}(a_1,\ldots,a_n)$ and $\varphi^{\underline{A}}(a_1,\ldots,a_n)$ are well-defined, because every variable in t (every free variable in φ, respectively) must appear in the extension.
- An extended term $t(x_1,\ldots,x_n)$ defines a function $t^{\underline{A}} : A^n \to A$, and an extended formula $\varphi(x_1,\ldots,x_n)$ defines a function $\varphi^{\underline{A}} : A^n \to \{\bot,\top\}$ in a structure $\underline{A}$ with universe A. In analogy to (unextended) formulas we write $\underline{A} \models \varphi(a_1,\ldots,a_n)$ if $\varphi^{\underline{A}}(a_1,\ldots,a_n) = \top$.
- An extended formula $\varphi(x_1,\ldots,x_n)$ defines a set of n-tuples from the universe A:

  $$\left\{(a_1,\ldots,a_n) \in A^n \mid \varphi^{\underline{A}}(a_1,\ldots,a_n) = \top\right\}$$

  In the following, we refer to this set as "the set defined by φ" if the structure $\underline{A}$ and the extension $(x_1,\ldots,x_n)$ are clear from the context.

The notation $\forall(\varphi)$ and $\exists(\varphi)$ is a short-hand for $\forall x_1 \cdots \forall x_n\,(\varphi)$ and $\exists x_1 \cdots \exists x_n\,(\varphi)$, where $\mathit{frvar}(\varphi) = \{x_1,\ldots,x_n\}$.

Definition 2.17 The (simultaneous) *substitution* $\theta := [x_1/t_1,\ldots,x_n/t_n]$ of the variables $x_1,\ldots,x_n$ by the terms $t_1,\ldots,t_n$ in a term is defined by:

(1) $y\theta := \begin{cases} t_i & \text{if } y = x_i \text{ for some } i \in \{1,\ldots,n\} \\ y & \text{otherwise} \end{cases} \quad \text{for } y \in \mathcal{V}$

(2) $f\theta := f$ for $f \in \mathcal{F}$

(3) $f(s_1,\ldots,s_m)\theta := f(s_1\theta,\ldots,s_m\theta)$ for $f \in \mathcal{F}$, $s_1,\ldots,s_m \in Tm(\mathcal{V},\Sigma)$

The (simultaneous) *substitution* $\theta := [x_1/t_1,\ldots,x_n/t_n]$ of the variables $x_1,\ldots,x_n$ by the terms $t_1,\ldots,t_n$ in a formula is defined by:

(1) $(s_1 = s_2)\theta := (s_1\theta = s_2\theta)$ for $s_1,s_2 \in Tm(\mathcal{V},\Sigma)$

(2) $r\theta := r$ for $r \in \mathcal{R}$

(3) $r(s_1,\ldots,s_m)\theta := r(s_1\theta,\ldots,s_m\theta)$ for $r \in \mathcal{R}$, $s_1,\ldots,s_m \in Tm(\mathcal{V},\Sigma)$

(4) $(\neg\varphi)\theta := (\neg\varphi\theta)$

(5) $(\varphi\,\rho\,\psi)\theta := (\varphi\theta\,\rho\,\psi\theta)$ for $\rho \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$

(6) $(Q\,y\,\varphi)\theta := \begin{cases} Q\,y\,(\varphi\theta'_i) & \text{if } y = x_i \text{ for some } i \in \{1,\dots,n\} \\ Q\,y\,(\varphi\theta) & \text{otherwise} \end{cases}$
for $Q \in \{\exists, \forall\}$, $y \in \mathcal{V}$, and $\theta'_i = [x_1/t_1, \dots, x_{i-1}/t_{i-1}, x_{i+1}/t_{i+1}, \dots, x_n/t_n]$

2.3 Algebra

2.3.1 Rings

Definition 2.18 A *commutative ring with* 1 (in the following "ring" for short) is a Σ_{ord}-structure $\underline{R}$ with universe R and the following properties:

Commutativity:	$\forall x \forall y (x + y = y + x)$ $\forall x \forall y (x \cdot y = y \cdot x)$
Associativity:	$\forall x \forall y \forall z \big((x+y)+z = x+(y+z)\big)$ $\forall x \forall y \forall z \big((x \cdot y) \cdot z = x \cdot (y \cdot z)\big)$
Distributivity:	$\forall x \forall y \forall z \big(x \cdot (y+z) = x \cdot y + x \cdot z\big)$
Identity:	$\forall x (x + 0 = x)$ $\forall x (x \cdot 1 = x)$
Additive Inverse:	$\forall x \big(x + (-x) = 0\big)$
	$\neg(0 = 1)$

A ring $\underline{R}$ is called a *domain* if it has no zero divisors.

No Zero-Divisors: $\forall x \forall y \big(x \cdot y = 0 \rightarrow (x = 0 \vee y = 0)\big)$

A ring $\underline{R}$ is called a *field* if multiplicative inverses exist for every non-zero element.

Multiplicative Inverse: $\forall x \big(\neg(x = 0) \rightarrow \exists y (x \cdot y = 1)\big)$

A domain or field $\underline{R}$ is called *ordered*, if

Ordering: $\forall x \forall y \forall z (x < y \rightarrow x + z < y + z)$
$\forall x \forall y (0 < x \wedge 0 < y \rightarrow 0 < x \cdot y)$

hold and $<$ is a strict linear order on R.

We deal mostly with the ordered field $\underline{\mathbb{R}}$ of real numbers (cf. Section 2.3.2) and its sub-fields of rational numbers $\underline{\mathbb{Q}}$ and algebraic numbers $\underline{\mathbb{A}}$ (cf. Section 2.3.5). In addition, we deal with polynomial rings (see Section 2.3.3).

2.3.2 The Real Numbers as Σ_{ord}-Structure

The structure $\underline{\mathbb{R}}$ is a Σ_{ord}-structure with the domain $\mathbb{R}$ and the usual operations and relations. That is, the operations $+^{\mathbb{R}}$, $-^{\mathbb{R}}$, $\cdot^{\mathbb{R}}$, $0^{\mathbb{R}}$, and $1^{\mathbb{R}}$ are the usual addition, negation, multiplication, zero, and one of the real numbers, and $<^{\mathbb{R}}$ is the usual less-than relation. It is an ordered field (Definition 2.18). We introduce some abbreviations.

The signature Σ_{ord} only contains the relation symbol $<$, so we can define:

$$
\begin{aligned}
(t_1 \leq t_2) &:= (t_1 < t_2 \vee t_1 = t_2)\\
(t_1 \neq t_2) &:= (t_1 < t_2 \vee t_2 < t_1)\\
(t_1 \geq t_2) &:= (t_2 < t_1 \vee t_2 = t_1)\\
(t_1 > t_2) &:= (t_2 < t_1)
\end{aligned}
$$

Note that the definitions of these abbreviations have been chosen so as not to contain negation, since virtual substitution requires positive formulas (cf. Definition 3.6 and Lemma 3.7).

To make the following presentation easier, we treat any formula $t_1\ \rho\ t_2$ with $\rho \in \{=,\neq,<,\leq,\geq,>\}$ as an atomic formula, although $\leq,\geq,>,\neq$ are not relation symbols from Σ_{ord}. The above abbreviations show that this does not introduce hidden negations into the formulas, which is important for algorithms which rely on positive formulas as input.

In addition, we assume that every atomic formula is of the form $t\ \rho\ 0$ for $\rho \in \{=,\neq,<,\leq,\geq,>\}$, although we may still write $t_1\ \rho\ t_2$ when appropriate. We call this the canonical form of atomic formulas in $\underline{\mathbb{R}}$. This requirement is no restriction since $t_1\ \rho\ t_2$ is equivalent to $(t_1 - t_2)\ \rho\ 0$ in $\underline{\mathbb{R}}$. When we speak of the "terms of a formula" ψ, we assume that the atomic formulas in ψ are in the canonical form $t\ \rho\ 0$ and mean the left-hand terms t. We also write integral numbers like 3 or -2 to denote the formal terms $1+1+1$ and $-(1+1)$, respectively.

The techniques described in Chapter 3 work in the reals, with the exception of root isolation (see Sections 2.3.4 and 3.4, which can also be used to compute exactly the integral solutions). The problems we try to solve are usually problems in the integers, e.g., enumerating the integral points in a polyhedron (cf. Section 2.1.3). We lift integral problems to the reals before applying our techniques. This implies that we will sometimes get the answer that some (real) solutions exist, although no integral solution exists. For Fourier-Motzkin elimination this has already been discussed in Section 2.1.3. The final results we compute may contain some superfluous cases (for non-integral parameter values, for example), but this does not affect the correctness of the results. It has turned out that we need not apply any special, integral methods to solve the problems we handle with the algorithms we develop in Chapter 4 and apply in Chapter 5.

The input for the problems will always be rational in the sense that all numbers appearing in the input are rational (or integral) numbers. This ensures that all formulas in the input can be expressed without denominators (by multiplying the formulas with common denominators); therefore Σ_{ord} is adequate for expressing the problems (see also Section 4.1).

2.3.3 Polynomial Rings

We assume that the reader is familiar with polynomial rings, so we do not introduce polynomial rings or the concepts necessary for dealing with them. A rigorous definition of polynomial rings via monoid rings is given in Chapter 2 of [BW93].

Definition 2.19 Let $\underline{R}$ be a domain with universe R and let $x_1,\ldots,x_n$ be some new symbols (called *indeterminates*). We define the set of *monomials* M

$$M := \{c \cdot x_1^{e_1} \cdot \cdots \cdot x_n^{e_n} \mid c \in R, e_1,\ldots,e_n \in \mathbb{N}\}$$

where, as usual, $x_i^0 := 1$. The polynomial ring $\underline{R}[x_1,\dots,x_n]$ over $\underline{R}$ is then defined by the universe

$$R[x_1,\dots,x_n] := \{m_1+\dots+m_k \mid k \in \mathbb{N}, m_1,\dots,m_k \in M\}$$

and the usual polynomial operations for $+$, $-$, $\cdot$.
If $n = 1$, the polynomial ring and the polynomials in it are called *univariate*; if $n \geq 2$ they are called *multivariate*.

According to this definition, a monomial is a multiple of a power product of the indeterminates with a coefficient from the underlying ring R. As is shown formally in [BW93], the ring axioms are inherited by $\underline{R}[x_1,\dots,x_n]$, and $\underline{R}[x_1,\dots,x_n]$ is again a domain.

When we are working in $\underline{\mathbb{R}}$, we can identify the terms $Tm(\{x_1,\dots,x_n\},\Sigma_{ord})$ with the elements of the polynomial ring $\underline{\mathbb{Z}}[x_1,\dots,x_n]$: since associativity and commutativity hold in $\underline{\mathbb{R}}$, every term in $Tm(\{x_1,\dots,x_n\},\Sigma_{ord})$ can be written as a sum of monomials.

Definition 2.20 Let $\underline{R}$ be a ring and $f = \sum_{i=0}^{n} c_i x^i \in R[x]$ with $c_n \neq 0$. Then we call

$\deg(f) := n$	the *degree* of f,
$\mathrm{lc}(f) := c_n$	the *leading coefficient* of f,

and for every $f = \sum_{i=0}^{n} c_i x^i \in R[x]$ we call

$$f' := \sum_{i=1}^{n} i c_i x^{i-1} \qquad \text{the } \textit{formal derivative} \text{ of } f.$$

Note that the degree and leading coefficient of the 0-polynomial are undefined.

We need polynomial division and greatest common divisors of polynomials. We cite here some results form Chapter 2 of [BW93]. We assume that the reader is familiar with polynomial division and the Euclidean algorithm (Chapter 2 of [BW93] contains these and many others).

Lemma 2.21 *(Polynomial division)*
Let $\underline{R}$ be a field and $f,g \in R[x]$ with $g \neq 0$. Then there exist uniquely determined $q,r \in R[x]$ with $f = qg + r$ and $\deg(r) < \deg(g)$ *or $r = 0$. q is called the quotient* $\mathrm{QUOT}(f,g)$ *of f and g, and r is called the remainder* $\mathrm{REM}(f,g)$ *of f and g.*

If $\mathrm{REM}(f,g) = 0$, we also write $\frac{f}{g}$ for $\mathrm{QUOT}(f,g)$.

Definition 2.22 Let $\underline{R}$ be a field and $a,b \in R[x]$, $a \neq 0$ or $b \neq 0$. Then $g \in R[x] \setminus \{0\}$ is called the *greatest common divisor* $\gcd(a,b)$ of a and b if

- $\mathrm{lc}(g) = 1$,
- $\mathrm{REM}(a,g) = \mathrm{REM}(b,g) = 0$,
- every $g' \in R[x] \setminus \{0\}$ with $\mathrm{REM}(a,g') = \mathrm{REM}(b,g') = 0$ satisfies $\mathrm{REM}(g,g') = 0$.

If $\gcd(a,b) = 1$, f and g are called *relatively prime*.
$\gcd(a,b)$ can be computed using the Euclidean algorithm.

The definition of the greatest common divisor can be extended from univariate polynomial rings to multivariate polynomial rings (cf. Section 2.5 of [BW93]) using the quotient fields of polynomial rings (Definition 2.28).

Squarefree Part of a Real Polynomial

Definition 2.23 Let $f \in \mathbb{R}[x] \setminus \{0\}$ with $\deg(f) \geq 1$.

$$f^* := \frac{f}{\gcd(f, f')}$$

is called the *squarefree part* of f.

The importance of the squarefree part f^* is that is has the same zeros as f, but every zero of f^* is simple, that is, if z is a zero of f^* then the linear factor $(x-z)$ divides f^*, but $(x-z)^2$ does not. This implies that the sign changes around the zero z and this property is useful for the root isolation described in Section 2.3.4. We note that one can also compute the *squarefree factorization* of a polynomial f

$$f = u \cdot f_1 \cdot f_2^2 \cdot f_3^3 \cdot \ldots \cdot f_n^n$$

where $u \in \mathbb{R}$ and $f_1, \ldots, f_n \in \mathbb{R}[x]$ are squarefree and pairwise relatively prime. An algorithm to compute a squarefree factorization can be found in Chapter 2.6 of [BW93]. The squarefree part and squarefree factorization of f have the advantage over f that the zeros of f^* and $f_1, \ldots, f_n$ are simple and that their degree is probably lower than the degree of f.

Reducta of Polynomials For cylindrical algebraic decomposition (Section 3.3) we need to define the reducta set of a polynomial, i.e., the polynomials obtained from a given polynomial f by successively removing the leading monomials. For example, consider the polynomial $f = 4x^3 + 3x^2 + 2x + 1$. The leading monomial of f is $4x^3$ and the reductum of f (written $\mathrm{red}(f)$) is $3x^2 + 2x + 1$. The second reductum $red^2(f) = \mathrm{red}\big(\mathrm{red}(f)\big)$ is $2x + 1$, and so forth. The reducta set of f (written $\mathrm{RED}(f)$) is the set of all the reducta obtained from f (including f itself); in this case $\mathrm{RED}(f) = \{4x^3 + 3x^2 + 2x + 1, 3x^2 + 2x + 1, 2x + 1, 1\}$.

Definition 2.24 Let $\underline{R}$ be a ring and $f \in R[x]$. We define the *reductum* of f as

$$\mathrm{red}(f) := \begin{cases} 0 & \text{if } f = 0 \\ f - \mathrm{lc}(f) \cdot x^{\deg(f)} & \text{otherwise} \end{cases}$$

The *reducta set* of f is defined as

$$\mathrm{RED}(f) := \big\{ red^i(f) \mid 0 \leq i \leq \deg(f), \mathrm{red}^i(f) \neq 0 \big\}$$

where $\mathrm{red}^i(f)$ denotes the functional iteration of red on f:
$\mathrm{red}^0(f) = f$, $\mathrm{red}^{i+1}(f) = \mathrm{red}^i\big(\mathrm{red}(f)\big)$.

Resultants and Subresultants Resultants play an important role in the arithmetic of algebraic numbers (cf. Section 2.3.5) and in the projection operator of cylindrical algebraic decomposition (cf. Section 3.3.2). They are defined as the determinants of certain matrices constructed from the coefficients of two given polynomials. Resultants are connected to the Euclidean algorithm: the resultant of two polynomials f and g vanishes if and only if $\deg\big(\gcd(f, g)\big) > 0$. The subresultants of f and g indicate whether polynomials of certain degrees appear as remainders during the Euclidean algorithm. We use resultants and subresultants in algebraic number calculus and cylindrical algebraic decomposition, but we are not discussing (sub-)resultants themselves here. Before we give their definitions, we look at an informal definition an give an example.

Let us look at the polynomials $f = 1x^2 + 2x + 3$ and $g = 4x^3 + 5x^2 + 6x + 7$. The resultant of f and g is defined as the determinant of the Sylvester matrix $S_0(f,g)$ of f and g:

$$S_0(f,g) = \begin{pmatrix} 1 & & & 4 & \\ 2 & 1 & & 5 & 4 \\ 3 & 2 & 1 & 6 & 5 \\ & 3 & 2 & 7 & 6 \\ & & 3 & & 7 \end{pmatrix}$$

As one can see, the Sylvester matrix consists of the coefficients of f and g which are written from top to bottom and repeated in a diagonal manner. The coefficients of f are repeated $\deg(g)$ times, and the coefficients of g are repeated $\deg(f)$ times. The determinant of the Sylvester matrix is called the resultant $\operatorname{res}(f,g)$ of f and g:

$$\operatorname{res}(f,g) = \det S_0(f,g) = 256$$

The subresultants are also determinants of matrices in the coefficients of f and g. The k-th subresultant of f and g is defined as the determinant of the matrix $S_k(f,g)$, where $S_k(f,g)$ is obtained from S_0 by deleting the last $2k$ rows, the last k columns with coefficients of f, and the last k columns with coefficients of g. Thus,

$$\begin{aligned} S_1(f,g) &= \begin{pmatrix} 1 & & 4 \\ 2 & 1 & 5 \\ 3 & 2 & 6 \end{pmatrix} \\ S_2(f,g) &= \begin{pmatrix} 1 \end{pmatrix} \end{aligned}$$

and the subresultants of f and g are

$$\begin{aligned} \operatorname{res}_1(f,g) &= \det S_1(f,g) = 0 \\ \operatorname{res}_2(f,g) &= \det S_2(f,g) = 1 \end{aligned}$$

We now cite the formal definition of (sub-)resultants from [vzGL02].

Definition 2.25 Let $\underline{R}$ be a ring and $f = \sum_{i=0}^{n} f_i x^i \in R[x] \setminus \{0\}$ and $g = \sum_{i=0}^{m} g_i x^i \in R[x] \setminus \{0\}$ with $\deg(f) = n \geq \deg(g) = m$. Then we define for $0 \leq k \leq m$ the $(n+m-2k) \times (n+m-2k)$ matrix $S_k(f,g)$ as

$$S_k(f,g) := \left(\begin{array}{cccc|cccccc} f_n & & & & g_m & & & & & \\ f_{n-1} & f_n & & & g_{m-1} & g_m & & & & \\ \vdots & & \ddots & & \vdots & & \ddots & & & \\ f_{n-m+k+1} & \cdots & \cdots & f_n & g_{k+1} & \cdots & \cdots & g_m & & \\ \vdots & & & \vdots & \vdots & & & & \ddots & \\ f_{k+1} & \cdots & \cdots & f_m & g_{m-n+k+1} & \cdots & \cdots & \cdots & \cdots & g_m \\ \vdots & & & \vdots & \vdots & & & & & \vdots \\ \vdots & & & \vdots & \vdots & & & & & \vdots \\ f_{2k-m+1} & \cdots & \cdots & f_k & g_{2k-n+1} & \cdots & \cdots & \cdots & \cdots & g_k \end{array} \right)$$

$$\qquad\qquad \underbrace{\hphantom{f_{2k-m+1} \cdots \cdots f_k}}_{m-k} \; \underbrace{\hphantom{g_{2k-n+1} \cdots \cdots \cdots \cdots g_k}}_{n-k}$$

where $f_j = 0$ and $g_j = 0$ for $j < 0$. Note that we assume $n \geq m$ in the definition of $S_k(f,g)$ just to be able to depict the matrix $S_k(f,g)$ easily; the definition of $S_k(f,g)$ extends naturally to the case $n < m$.

The matrix $S_0(f,g)$ is called the *Sylvester* matrix of f and g.

Definition 2.26 Let $\underline{R}$ be a ring and $f,g \in R[x] \setminus \{0\}$. We call

$$\mathrm{res}_k(f,g) := \det S_k(f,g)$$

the *k-th subresultant* of f and g for $0 \leq k \leq \min\{\deg(f), \deg(g)\}$.
$\mathrm{res}(f,g) := \mathrm{res}_0(f,g)$ is also called the *resultant* of f and g.

Definition 2.27 Let $\underline{R}$ be a ring and $f,g \in R[x] \setminus \{0\}$. We call

$$\mathrm{PSC}(f,g) := \{\mathrm{res}_k(f,g) \mid 0 \leq k \leq \min\{\deg(f), \deg(g)\},\ \mathrm{res}_k(f,g) \neq 0\}$$

the *PSC set* of f and g. We additionally define $\mathrm{PSC}(f,g) = \varnothing$ if $f = 0$ or $g = 0$.

For the above (sub-)resultant example the PSC set is $\mathrm{PSC}(f,g) = \{\mathrm{res}_0(f,g), \mathrm{res}_2(f,g)\} = \{256, 1\}$, since $\mathrm{res}_1(f,g) = 0$.

We note that we use the term “PSC set” here to be consistent with [ACM98]. The abbreviation “psc” denotes the “principal subresultant coefficient”, i.e. the coefficient of x^j in the j-th *polynomial* subresultant of f and g. Our Definition 2.26 uses *scalar* subresultants (which are exactly the principal subresultant coefficients). See [vzGL02] for detailed descriptions of scalar and polynomial subresultants.

Quotient Field of a Polynomial Ring Finally, we introduce briefly the quotient field of a polynomial ring since in the generalized polyhedron model the coefficients of variables are usually taken from $\mathbb{Q}(p_1, \ldots, p_m)$ with $p_1, \ldots, p_m$ being the parameters (cf. Section 4.1).

Definition 2.28 Given a domain $\underline{R}$ with universe R, we define the field $\underline{R}(x_1, \ldots, x_n)$ with universe $R(x_1, \ldots, x_n)$ through

$$R(x_1, \ldots, x_n) := \{\frac{f}{g} \mid f,g \in R[x_1, \ldots, x_n], g \neq 0\}$$

The arithmetic in $\underline{R}(x_1, \ldots, x_n)$ is carried out like the usual arithmetic with fractions. The multiplicative inverse of a fraction $\frac{f}{g} \in R(x_1, \ldots, x_n) \setminus \{0\}$ is defined as

$$\left(\frac{f}{g}\right)^{-1} := \frac{g}{f}$$

The details can be found in [BW93].
We can embed $R[x_1, \ldots, x_n]$ into $R(x_1, \ldots, x_n)$ by

$$f \mapsto \frac{f}{1}$$

and therefore treat $R[x_1, \ldots, x_n]$ as sub-ring of $R(x_1, \ldots, x_n)$.

2.3.4 Real Roots of Polynomials

In the following we look at polynomials $f \in \mathbb{R}[x]$ and discuss how to find their real zeros, i.e., real numbers $r \in \mathbb{R}$ such that $f(r) = 0$.

Definition 2.29 Let $f \in \mathbb{R}[x] \setminus \{0\}$ and $r \in \mathbb{R}$ with $f(r) = 0$. An interval $[a,b]$ with $a,b \in \mathbb{Q}$ is called an *isolation interval* for the zero r of f, if $r \in [a,b]$ and r is the only zero of f in $[a,b]$ (i.e., $f(c) \neq 0$ for every $c \in [a,b] \setminus \{r\}$).

Our aim is to find isolation intervals for every root of a given polynomial $f \in \mathbb{R}[x] \setminus \{0\}$ such that the intervals are pairwise disjoint. We achieve this goal using a simple method: we start with an interval containing every zero of f and successively divide the interval in two equally-sized subintervals until every interval contains exactly one zero. This requires two other algorithms: an algorithm to find the initial interval which includes every zero of f, and an algorithm to count the number of (distinct) zeros of f in a given interval (to find out when we are finished dividing the intervals).

Lemma 2.30 *([BW93], Exercise 8.114)*
Let $f = \sum_{i=0}^{n} c_i x^i \in \mathbb{R}[x] \setminus \{0\}$ with $\deg(f) \geq 1$ and $r \in \mathbb{R}$ with $f(r) = 0$. Then

$$|r| \leq 1 + \frac{1}{|c_n|} \max\{|c_0|, \ldots, |c_{n-1}|\}$$

Proof. The proposition is obviously true for $|r| \leq 1$. Let $|r| > 1$. Since $c_n r^n = -\sum_{i=1}^{n-1} c_i r^i$, we have

$$\begin{aligned}
|c_n| \cdot |r|^n = |\sum_{i=0}^{n-1} c_i r^i| &\leq \sum_{i=0}^{n-1} |c_i| \cdot |r|^i \\
&\leq \max\{|c_0|, \ldots, |c_{n-1}|\} \cdot \sum_{i=0}^{n-1} |r|^i = \max\{|c_0|, \ldots, |c_{n-1}|\} \cdot \frac{|r|^n - 1}{|r| - 1} \\
&\leq \max\{|c_0|, \ldots, |c_{n-1}|\} \cdot \frac{|r|^n}{|r| - 1}
\end{aligned}$$

From this we can deduce $|r| \leq 1 + \frac{1}{|c_n|} \max\{|c_0|, \ldots, |c_{n-1}|\}$. □

The following theorem shows how to count the number of distinct zeros in a given interval. It uses the notation $\text{VARSIGN}(a_1, \ldots, a_n)$ to denote the number of sign changes in the sequence $(a_1, \ldots, a_n)$. "Sign changes" refers to the number of changes in the signum of the values $a_1, \ldots, a_n$ where zeros are ignored. Thus, $\text{VARSIGN}(1, -2, 3, 4, -5)$ is 3: the signum changes between 1 and -2, between -2 and 3, and finally between 4 and -5. $\text{VARSIGN}(1, 0, -2, 3, 0, 4, 0, 0, -5, 0)$ is also 3 since zeros are ignored.

Theorem 2.31 *(Sturm's Theorem [BW93])*
Let $f \in \mathbb{R}[x] \setminus \{0\}$ with $\deg(f) \geq 1$ and $a, b \in \mathbb{R}$ with $a \leq b$ and $f(a), f(b) \neq 0$. Define

$$\begin{aligned}
f_0 &:= f \\
f_1 &:= f' \\
f_{i+1} &:= -\text{REM}(f_{i-1}, f_i) && \textit{as long as } f_i \neq 0
\end{aligned}$$

Let $r \in \mathbb{N}$ be maximal such that $f_r \neq 0$. Then f has the following number of distinct zeros in the interval $[a,b]$:

$$\mathrm{VARSIGN}\big(f_0(a),\ldots,f_r(a)\big) - \mathrm{VARSIGN}\big(f_0(b),\ldots,f_r(b)\big)$$

A proof of Sturm's theorem can be found in [BW93], Chapter 8.8.

Using the preceding lemma and Sturm's theorem, it is possible to find isolation intervals for every zero of a given polynomial $f \in \mathbb{R}[x] \setminus \{0\}$:

(1) Using Lemma 2.30, we compute $M > 0$ such that every root of f lies in $[a,b]$ where $a = -M$ and $b = M$.

(2) If a is a zero of f, we divide f as often as possible by $x-a$ (to eliminate the factor $x-a$ from f) and $[a,a]$ is an isolation interval for the zero a of f; the same takes place with $x-b$ as factor and $[b,b]$ as isolation interval, if b is a zero of f. Let the resulting polynomial be g.

(3) If the number of zeros of g in $]a,b[$ is 0, we terminate with "no zeros in $[a,b]$."

(4) If the number of zeros of g in $]a,b[$ is 1, we terminate with the isolation interval $[a,b]$.

(5) If there is more than one zero of g in $]a,b[$, we set $c := \frac{1}{2}(a+b)$ and apply the algorithm recursively, starting with the intervals $[a,c]$ and $[c,b]$ for the polynomial g at Step (2).

The intervals computed by the above algorithm are not disjoint in general: some interval endpoints may be the identical. However, by the following lemma, it is possible to shrink isolation intervals to become arbitrarily small.

Lemma 2.32 *Let $f \in \mathbb{R}[x] \setminus \{0\}$ with $\deg(f) \geq 1$ and $[a,b]$ an isolation interval for a zero r of f and $c = \frac{1}{2}(a+b)$. Then $[a,c]$ or $[c,c]$ or $[c,b]$ is an isolation interval for r.*

This lemma is pretty obvious. Either c is the only zero of f in $[a,b]$, or the zero is not c and lies inside one of the intervals $[a,c]$ or $[c,b]$. Which of the two intervals $[a,c]$ and $[c,b]$ the zero lies in can be decided by counting the zeros in each interval using Sturm's theorem. But there is a computationally more efficient alternative using the squarefree part f^* of f: since f—and hence f^*—has exactly one zero in $[a,b]$ and that zero is a single root of f^*, it follows that the signs of $f^*(a)$ and $f^*(b)$ are different, i.e., $f^*(a) \cdot f^*(b) < 0$. The sign of $f^*(c)$ determines which of the two intervals is the right one: if $f^*(a) \cdot f^*(c) < 0$ then $[a,c]$ is the right interval, otherwise it is $[c,b]$. Using Lemma 2.32 one can make the isolation intervals found by the preceding algorithm smaller and smaller until they are pairwise disjoint.

2.3.5 Algebraic Numbers

The previous section shows that one can find isolation intervals with rational bounds for every zero of polynomials $f \in \mathbb{Q}[x]$. This means that it is possible to describe any such zero solely by rational numbers (the coefficients of f and the interval bounds).

Definition 2.33 Let $\mathbb{A} \subset \mathbb{R}$ be the set of all zeros of polynomials over $\mathbb{Q}$, i.e.,

$$\mathbb{A} := \{r \in \mathbb{R} \mid \text{there exists } f \in \mathbb{Q}[x] \text{ with } f(r) = 0\}$$

We call $\mathbb{A}$ the set of *algebraic numbers*.

$\mathbb{A}$ is a superset of $\mathbb{Q}$, since every rational number $q \in \mathbb{Q}$ is the zero of $(x-q) \in \mathbb{Q}[x]$. It is possible to do arithmetic in $\mathbb{A}$, that is, one can define addition, negation, multiplication, and reciprocal values of algebraic numbers and they are again algebraic numbers.

Theorem 2.34 *The algebraic numbers $\mathbb{A}$ form a field.*

Proof: [Loo83].

The arithmetic computations in $\mathbb{A}$ are, in principle, always performed by the following steps:

- From the given algebraic numbers (given by their polynomials and isolation intervals) compute a polynomial s which has the result of the arithmetic operation among its zeros.
- Compute pairwise disjoint isolation intervals $I_1, \ldots, I_n$ for the zeros of s.
- From the given isolation intervals of the input numbers compute (by interval arithmetic) an interval K which contains the resulting algebraic number.
- If $K \cap I_i \neq \varnothing$ for more than one i, shrink the isolation intervals of the input numbers and compute a new, smaller interval K from them. Repeat this until K has a non-empty intersection with exactly one I_i.
- The polynomial s and the interval K describe the algebraic number resulting from the arithmetic operation.

As an example consider the two algebraic numbers $\alpha = \sqrt{2}$ and $\beta = \sqrt{3}$. The product $\alpha \cdot \beta$ is the algebraic number $\sqrt{6}$.

We first cite the algorithm for addition and multiplication of algebraic numbers from [Loo83] and then show exemplarily the computation of $\sqrt{2} \cdot \sqrt{3}$.

The notation $\mathrm{res}_y\big(f(x,y), g(x,y)\big)$ is used below to express that we want to compute the resultant of $f(x,y)$ and $g(x,y)$ where f and g are to be treated as polynomials in y (having coefficients from $\mathbb{Q}[x]$), i.e., the entries of the Sylvester matrix (cf. Definition 2.25) are polynomials in x, and hence, the resultant is also a polynomial in x.

- Input: Two algebraic numbers α and β described by their polynomials a and b and by their isolation intervals I and J.
- Compute:
  - for addition: $s(x) = \mathrm{res}_y\big(a(x-y), b(y)\big)$
    for multiplication: $s(x) = \mathrm{res}_y\big(y^{\deg(a)} \cdot a(x/y), b(y)\big)$
  - squarefree factorization of s: $s(x) = u \cdot d_1(x) \cdot d_2(x)^2 \cdot \ldots \cdot d_f(x)^f$
  - pairwise disjoint isolation intervals $I_1, \ldots, I_n$ for every zero of every d_i $(1 \leq i \leq f)$
  - an interval which contains $\alpha + \beta$ (or $\alpha \cdot \beta$, respectively); for addition: $K = I + J$, for multiplication $K = I * J$ (using interval arithmetic)
  - if there is more than one I_i with $K \cap I_i \neq \varnothing$, then bisect I and J (using Lemma 2.32) and go back to the previous step
- Output: the polynomial d from $d_1, \ldots, d_n$ which has a zero in I_i and the isolation interval K together describe the algebraic number $\gamma = \alpha + \beta$ (or $\gamma = \alpha \cdot \beta$).

For the example $\alpha = \sqrt{2}$, $\beta = \sqrt{3}$ we use the defining polynomials $a(x) = x^2 - 2$ and $b(x) = x^2 - 3$ and the isolation interval $[0,2]$ for both α and β. We obtain $\deg(a) = 2$ and need to calculate (for the multiplication of α and β) $y^2 \cdot a(\frac{x}{y}) = -2 \cdot y^2 + 0 \cdot y + x^2$ and $b(y) = 1 \cdot y^2 + 0 \cdot y - 3$. From these polynomials we calculate the resultant $s(x)$; in the depicted matrix we only show the entries coming from the coefficients of $y^2 \cdot a(\frac{x}{y})$ and $b(y)$, the other entries are 0 by Definition 2.25:

$$s(x) = \mathrm{res}_y\big(y^2 \cdot a(\frac{x}{y}), b(y)\big) = \det \begin{pmatrix} -2 & & 1 & \\ 0 & -2 & 0 & 1 \\ x^2 & 0 & -3 & 0 \\ & x^2 & & -3 \end{pmatrix} = x^4 - 12x^2 + 36$$

A squarefree factorization of $s(x)$ is $\big(x^2 - 6\big)^2$. The zeros of $s(x)$ are therefore $-\sqrt{6}$ and $\sqrt{6}$, and we choose $[-3,-2]$ and $[2,3]$ as isolation intervals for these zeros. The interval K is now computed from the isolation intervals of α and β, and in this case we can use $K = [0 \cdot 0, 2 \cdot 2] = [0,4]$. Only the interval $[2,3]$ has a non-empty intersection with the interval K and hence, the algebraic number $\alpha \cdot \beta$ is defined by the polynomial $d(x) = x^2 - 6$ and the isolation interval $K = [0,4]$ (we can also use $[2,3]$, of course). Obviously, $d(x)$ and K describe the algebraic number $\sqrt{6}$, which is the product of α and β.

Algorithms SIMPLE and NORMAL For the cylindrical algebraic decomposition method of Section 3.3 we need two algorithms from algebraic number calculus. We present the specification of these algorithms as lemmas here and refer the reader to [Loo83] for the concrete algorithms and proofs of their correctness.

Lemma 2.35 *(*SIMPLE*)*
Let $\alpha, \beta \in \mathbb{A}$. *The algorithm* SIMPLE *computes* $\gamma \in \mathbb{A}$ *and* $a, b \in \mathbb{Q}[x]$ *such that* $\alpha = a(\gamma)$ *and* $\beta = b(\gamma)$.

As an example for this lemma, we look at $\alpha = \sqrt{2}$ and $\beta = \sqrt[3]{2} + 1$. A possible output of the algorithm SIMPLE is $\gamma = \sqrt[6]{2}$ and $a(x) = x^3$, $b(x) = x^2 + 1$, since $a(\sqrt[6]{2}) = \sqrt[6]{2}^3 = \sqrt{2} = \alpha$ and $b(\sqrt[6]{3}) = \sqrt[6]{2}^2 + 1 = \sqrt[3]{2} + 1 = \beta$.

Lemma 2.36 *(*NORMAL*)*
Let $f \in \mathbb{A}[x]$. *The algorithm* NORMAL *computes* $g \in \mathbb{Q}[x]$ *such that every zero of f is also a zero of g, that is $f(r) = 0$ implies $g(r) = 0$ for every $r \in \mathbb{R}$.*

As an example consider the polynomial $f(x) = x - \sqrt{2} \in \mathbb{A}[x]$ with the algebraic number coefficient $\sqrt{2}$. The only zero of f is $\sqrt{2}$. The rational polynomial $x^2 - 2 \in \mathbb{Q}[x]$ has rational coefficients only and $\sqrt{2}$ is one of its zeros. The polynomial $x^2 - 2$ has another zero, namely $-\sqrt{2}$. This shows that the rational polynomial computed by NORMAL has, in general, more zeros than the original polynomial with algebraic number coefficients.

Chapter 3

Quantifier Elimination

Section 2.2 introduces quantifier-free formulas and the first-order quantifiers $\exists$ and $\forall$. The quantifiers are used to express properties of some variables by the use of other variables. For example, the formula $\exists y\,(x = y^2)$ makes a statement about the variable x by claiming the existence of a value (denoted by y) satisfying the condition $x = y^2$. If we are working in the real numbers, this obviously means that $x \geq 0$, since exactly the non-negative real numbers are squares of other real numbers. Both formulas, $\exists y\,(x = y^2)$ and $x \geq 0$, make the same statement about x (in the real numbers). The difference is that the latter can make this statement without the use of quantifiers.

In the real numbers it is always possible to replace a formula with quantifiers by an equivalent quantifier-free formula. In this chapter we show how to compute such quantifier-free equivalents for some special cases using the techniques called *virtual substitution* and *cylindrical algebraic decomposition*.

3.1 Definitions

Before we start with the concrete algorithms, we give the formal definition of quantifier elimination.

Definition 3.1 Let Σ be a signature. A Σ-structure $\underline{A}$ is said to allow *quantifier elimination*, if, for every formula $\varphi \in Fm(\mathcal{V}, \Sigma)$, a formula ψ satisfying the following conditions exists:

(1) $\psi \in Qf(\mathcal{V}, \Sigma)$

(2) $\underline{A} \models \varphi \leftrightarrow \psi$

(3) $frvar(\psi) \subseteq frvar(\varphi)$

Conditions (1) and (2) state that ψ is quantifier-free and logically equivalent to φ in the structure $\underline{A}$, and condition (3) requires that ψ only contains variables which occur freely in φ.

As stated above, the formula $\exists y\,(x = y^2)$ expresses that x is the square of some real number. In the structure $\underline{\mathbb{R}}$ this is equivalent to saying that x is non-negative: $x \geq 0$. Obviously all three conditions of Definition 3.1 hold:

(1) $(x \geq 0) \in Qf(\mathcal{V}, \Sigma_{ord})$

(2) $\underline{\mathbb{R}} \models \exists y\,(x = y^2) \leftrightarrow x \geq 0$

(3) $\mathit{frvar}(x \geq 0) = \{x\} \subseteq \{x\} = \mathit{frvar}\big(\exists y\,(x = y^2)\big)$.

Definition 3.2 A *decision method* for a structure $\underline{A}$ is an algorithm which, given a formula $\varphi \in Fm(\mathcal{V}, \Sigma)$ with $\mathit{frvar}(\varphi) = \varnothing$ as input, computes whether $\underline{A} \models \varphi$ or $\underline{A} \not\models \varphi$ holds.

A quantifier elimination method for a structure $\underline{A}$ yields a decision method under the additional condition that atomic formulas can be decided in $\underline{A}$. Given a formula φ with $\mathit{frvar}(\varphi) = \varnothing$, the quantifier elimination method calculates an equivalent formula ψ with $\mathit{frvar}(\psi) = \varnothing$. Whether $\underline{A} \models \psi$ or $\underline{A} \not\models \psi$ holds, depends only on whether the atomic formulas of ψ hold in $\underline{A}$ or not (cf. Definition 2.13). Hence, if the atomic formulas of ψ can be decided, the truth-value of ψ, which is also the truth-value of φ, can be computed.

Alfred Tarski proved in 1948 that the above quantifier elimination example in $\underline{\mathbb{R}}$ does not succeed just by chance.

Theorem 3.3 *(Tarski [Tar51])*
The structure $\underline{\mathbb{R}}$ allows quantifier elimination, and there is an effective algorithm to perform quantifier elimination on a given formula.

Since it is in principle possible to decide atomic formulas in $\underline{\mathbb{R}}$, this theorem also proves the existence of a decision method for $\underline{\mathbb{R}}$. Since Tarski's discovery of quantifier elimination for $\underline{\mathbb{R}}$ many other (and more efficient) algorithms have been found. We present briefly quantifier elimination by *virtual substitution* (Section 3.2) and a decision method based on cylindrical algebraic decomposition (CAD, Section 3.3).

We should note here that no quantifier elimination procedure for Σ_{ord} exists in $\underline{\mathbb{Q}}$. In $\underline{\mathbb{R}}$ the formula $\exists y\,(x = y^2)$ is equivalent to $x \geq 0$. In $\underline{\mathbb{Q}}$ the condition $x \geq 0$ is necessary for x to be the square of a rational number, but it is not sufficient since, for example, $x = 2$ is not a square of a rational number. No quantifier-free formula with the only variable $x \in \mathbb{Q}$ can express that x is a square.

3.2 Virtual Substitution

Virtual substitution was discovered by Volker Weispfenning [Wei88]. It derives its name from the way it eliminates quantifiers. To get rid of an existential quantifier, it is replaced by a disjunction where in each disjunct the quantified variable is substituted (using the special virtual substitution) by appropriately chosen terms (cf. Definitions 3.13, 3.14 and Theorem 3.16).

To illustrate the idea, let us look at the very simple example:

$$\varphi := \exists x \psi$$
$$\psi := (x \geq 1 \wedge x \leq p)$$

φ expresses that there is a real number between 1 and p, so it is obviously equivalent to saying that $p \geq 1$. As we will see in Section 3.2.2 (Definition 3.15)—and as is probably intuitively clear—the "critical points" for x in the formula ψ are $x = 1$ and $x = p$. Virtual substitution now proposes that φ is equivalent to $\psi[x /\!\!/ 1] \vee \psi[x /\!\!/ p]$. Here $[\cdot /\!\!/ \cdot]$ denotes virtual substitution and, in this simple case, it is the same as usual substitution. Therefore ψ is equivalent to $(1 \geq 1 \wedge 1 \leq p) \vee (p \geq 1 \wedge p \leq p)$ which can be simplified to $p \geq 1$ and that is the result we expect.

3.2.1 Prerequisites

Virtual substitution itself cannot take every first-order formula as input for the quantifier elimination, but it requires the formula to be prenex with a positive matrix.

Definition 3.4 A formula $\varphi \in Fm(\mathcal{V},\Sigma)$ is called *prenex* if it is of the form

$$Q_1 x_1 \cdots Q_n x_n\ \psi$$

with $n \in \mathbb{N}$, $Q_1,\dots,Q_n \in \{\exists,\forall\}$, $x_1,\dots,x_n \in \mathcal{V}$, and $\psi \in Qf(\mathcal{V},\Sigma)$. ψ is called the *matrix* of the prenex formula φ.

Lemma 3.5 *Every formula $\varphi \in Fm(\mathcal{V},\Sigma)$ is equivalent to a prenex formula $\psi \in Fm(\mathcal{V},\Sigma)$.*

Proof sketch. A quantifier can always be moved to the front of a formula by applying the following transformations as often as necessary:
For every $\varphi \in Fm(\mathcal{V},\Sigma)$ and $x \in \mathcal{V}$
$\neg(\exists x\varphi)$ is equivalent to $\forall x(\neg\varphi)$ and $\neg(\forall x\varphi)$ is equivalent to $\exists x(\neg\varphi)$.
For every $\varphi,\psi \in Fm(\mathcal{V},\Sigma)$, $\rho \in \{\wedge,\vee,\rightarrow,\leftrightarrow\}$ and $x,y \in \mathcal{V}$ with $y \notin frvar(\varphi) \cup frvar(\psi)$
$\big(\varphi\,\rho\,(Qx\psi)\big)$ is equivalent to $Qy\,\big(\varphi\,\rho\,(\psi[x/y])\big)$, and
$\big((Qx\varphi)\,\rho\,\psi\big)$ is equivalent to $Qy\,\big((\varphi[x/y])\,\rho\,\psi\big)$
Repeating this process for every quantifier will finally yield a prenex formula. □

Definition 3.6 A formula $\varphi \in Qf(\mathcal{V},\Sigma)$ is called *positive* if none of the junctors $\neg$, $\rightarrow$, $\leftrightarrow$ appears in φ, i.e., φ only contains $\wedge$ and $\vee$ as junctors.

Lemma 3.7 *In the structure $\underline{\mathbb{R}}$ every formula $\varphi \in Qf(\mathcal{V},\Sigma_{ord})$ is equivalent to a positive formula $\psi \in Qf(\mathcal{V},\Sigma_{ord})$.*

Proof. Due to the definition of the semantics of $\rightarrow$ and $\leftrightarrow$ these junctors can be replaced: the formula $\sigma \rightarrow \tau$ is equivalent to $(\neg\sigma)\vee\tau$, and $\sigma \leftrightarrow \tau$ is equivalent to $(\sigma\wedge\tau)\vee(\neg\sigma\wedge\neg\tau)$. A formula with the junctors $\neg$, $\wedge$, and $\vee$ is made positive by "pushing" the negations into the formula until they reach atomic formulas. This proof is by induction on the structure of a formula $\varphi \in Qf(\mathcal{V},\Sigma_{ord})$ with the junctors $\neg$, $\wedge$, and $\vee$ only:

$\varphi \in At(\mathcal{V},\Sigma_{ord})$: φ is positive by definition.

$\varphi = \neg(t_1 < t_2)$: φ is equivalent to the positive formula $t_1 \geq t_2$ in $\underline{\mathbb{R}}$.

$\varphi = \neg(t_1 = t_2)$: φ is equivalent to the positive formula $t_1 \neq t_2$ in $\underline{\mathbb{R}}$.

$\varphi = \neg\neg\varphi_1$: By induction hypothesis, φ_1 is equivalent to a positive formula ψ, which is then also equivalent to φ.

$\varphi = \neg(\varphi_1\wedge\varphi_2)$: By induction hypotheses, there exist positive formulas $\psi_1,\psi_2 \in Qf(\mathcal{V},\Sigma_{ord})$ with ψ_1 is equivalent to $\neg\varphi_1$ and ψ_2 is equivalent to $\neg\varphi_2$. Then φ is equivalent to the positive formula $\psi_1\vee\psi_2$.

$\varphi = \neg(\varphi_1\vee\varphi_2)$: Analogous to $\neg(\varphi_1\wedge\varphi_2)$ with $\wedge$ and $\vee$ exchanged.

$\varphi = \varphi_1\wedge\varphi_2$: By induction hypotheses there exist positive formulas ψ_1 and ψ_2 where ψ_1 is equivalent to φ_1 and ψ_2 is equivalent to φ_2. Therefore φ is equivalent to $\psi_1\wedge\psi_2$.

$\varphi = \varphi_1 \vee \varphi_2$: Like $\varphi_1 \wedge \varphi_2$, just with $\vee$ instead of $\wedge$.

The termination of this recursive transformation process is ensured by the fact that the number of junctors in the formula decreases in every stop of the recursion. □

3.2.2 The Basic Algorithm for Linear Formulas

Let us look at the elimination of a single existential quantifier. The extension of the elimination technique to more than one quantifier (and universal quantifiers) is discussed in Section 3.2.4. The presentation of the method we give here is based on the lecture "Anwendungen der Computerlogik" given by Dr. Andreas Dolzmann in the summer semester 2002. A detailed description of the method (and proofs for lemmas which we leave out or only sketch here) can be found in [Wei88] and [LW93].

The case we look at is a formula $\exists x\,\psi$, where ψ is a formula which is linear in x, positive, and quantifier-free.

Definition 3.8 A formula ψ is called *linear in* $\{x_1,\ldots,x_n\} \subseteq \mathcal{V}$ if, for every variable $x \in \{x_1,\ldots,x_n\}$, every term appearing in ψ can be written in the form $t \cdot x + t'$ with $x_1,\ldots,x_n \notin var(t)$ and $x \notin var(t')$.

We assume that $frvar(\psi) \subseteq \{u_1,\ldots,u_m,x\}$ so that $(u_1,\ldots,u_m,x)$ is an extension of ψ in the sense of Definition 2.15. This implies that $(u_1,\ldots,u_m)$ is an extension for φ (since $x \notin frvar(\varphi)$). $u_1,\ldots,u_m$ are called *parameters*. In the following, we always consider the extension $(u_1,\ldots,u_m,x)$ for ψ.

An existential quantifier like in $\exists x\,\psi$ claims that an element of the universe (in our case a real number) exists which satisfies a certain condition. The idea of quantifier elimination with virtual substitution is to find a *finite* set of candidates for the quantified variable such that, if there are satisfying elements, one of the candidates is among them.

To find such finite sets of candidates (also called "test points"), let us take a closer look at the structure of the set of all real numbers satisfying the condition ψ, called the satisfaction set.

Definition 3.9 Let $(a_1,\ldots,a_n) = \bar{a} \in \mathbb{R}^n$. The *satisfaction set* $S^x_{\bar{a}}(\psi)$ of the formula ψ (with respect to its free variable x) is defined as

$$S^x_{\bar{a}}(\psi) := \{c \in \mathbb{R} \mid \psi^{\mathbb{R}}(\bar{a},c) = \top\}$$

Lemma 3.12 states that the satisfaction set of ψ is of a special form: a union of finitely many disjoint intervals.

Definition 3.10 $S \subseteq \mathbb{R}$ is called a *union of finitely many disjoint maximal intervals*, if $S = \dot{\bigcup}_{i=1}^{k} I_i$, $k \in \mathbb{N}$, $I_1,\ldots,I_k$ are pairwise disjoint intervals in $\mathbb{R}$, and for every $i = 1,\ldots,k$ and every interval J, $J \supsetneq I_i$: $J \nsubseteq S$.

Furthermore, we define the sets of weak lower bounds $B_{wl}(S)$, weak upper bounds

$B_{wu}(S)$, strict lower bounds $B_{sl}(S)$, and strict upper bounds $B_{su}(S)$ of the intervals $I_1,\dots,I_k$.

$$\begin{aligned}
B_{wl}(S) :=& \{b \in \mathbb{R} \mid i \in \{1,\dots,k\}, [b,b'] = I_i, b' \in \mathbb{R}\} \cup \\
& \{b \in \mathbb{R} \mid i \in \{1,\dots,k\}, [b,b'[= I_i, b' \in \mathbb{R} \cup \{\infty\}\} \\
B_{wu}(S) :=& \{b \in \mathbb{R} \mid i \in \{1,\dots,k\}, [b',b] = I_i, b' \in \mathbb{R}\} \cup \\
& \{b \in \mathbb{R} \mid i \in \{1,\dots,k\},]b',b] = I_i, b' \in \mathbb{R} \cup \{-\infty\}\} \\
B_{sl}(S) :=& \{b \in \mathbb{R} \mid i \in \{1,\dots,k\},]b,b'] = I_i, b' \in \mathbb{R}\} \cup \\
& \{b \in \mathbb{R} \mid i \in \{1,\dots,k\},]b,b'[= I_i, b' \in \mathbb{R} \cup \{\infty\}\} \\
B_{su}(S) :=& \{b \in \mathbb{R} \mid i \in \{1,\dots,k\}, [b',b[= I_i, b' \in \mathbb{R}\} \cup \\
& \{b \in \mathbb{R} \mid i \in \{1,\dots,k\},]b',b[= I_i, b' \in \mathbb{R} \cup \{-\infty\}\}
\end{aligned}$$

Lemma 3.11 *Let $S_1 = \dot\bigcup_{i=1}^k I_i$, $S_2 = \dot\bigcup_{i=1}^l J_i$ be unions of finitely many disjoint maximal intervals. Then*

(1) $S_1 \cup S_2$ and $S_1 \cap S_2$ are unions of finitely many disjoint maximal intervals.

(2) $B_\tau(S_1 \cup S_2) \subseteq B_\tau(S_1) \cup B_\tau(S_2)$ for $\tau \in \{wl, wu, sl, su\}$

(3) $B_\tau(S_1 \cap S_2) \subseteq B_\tau(S_1) \cup B_\tau(S_2)$ for $\tau \in \{wl, wu, sl, su\}$

Let us consider the formula $\alpha = \big((x \geq 0 \wedge 2x \leq p) \vee x > 7\big)$. The canonical form of this formula is $(x \geq 0 \wedge 2x - p \leq 0) \vee x - 7 > 0$. Obviously, the satisfaction set of α is $[0, \frac{p}{2}] \cup]7, \infty[$ (where we interpret $[0, \frac{p}{2}]$ as the empty set for $p < 0$). The finite bounds of the intervals in the satisfaction set are the solutions of the equations $x = 0$, $2x - p = 0$, and $x - 7 = 0$. The following lemma shows that we can always find the finite interval bounds of the satisfaction set of a formula ψ among the zeros of the terms in ψ. Of course, not every zero is an interval bound, since, for example, $\alpha \vee x \geq 8$ has the additional zero 8 (compared to α), but the interval $[8, \infty[$ is a subset of the interval $]7, \infty[$, so 8 does not appear as a bound in the satisfaction set.

Lemma 3.12 *The satisfaction set $S^x_{\bar a}(\psi)$ of the formula ψ is a finite union of pairwise disjoint maximal intervals. For every $\tau \in \{wl, wu, sl, su\}$ the finite bounds in $B_\tau\big(S^x_{\bar a}(\psi)\big)$ are a subset of the zeros of the terms in ψ.*

Proof sketch.

$\psi = \psi_1 \wedge \psi_2$:
: $S^x_{\bar a}(\psi_1 \wedge \psi_2) = S^x_{\bar a}(\psi_1) \cap S^x_{\bar a}(\psi_2)$ by definition of $S^x_{\bar a}$ and the semantics of $\wedge$. By the induction hypothesis, $S^x_{\bar a}(\psi_1) = \dot\bigcup_{i=1}^k I_i$ and $S^x_{\bar a}(\psi_2) = \dot\bigcup_{i=1}^l J_i$ are finite disjoint unions of maximal intervals and the finite bounds in $B_\tau\big(S^x_{\bar a}(\psi_1)\big)$, $B_\tau\big(S^x_{\bar a}(\psi_2)\big)$ ($\tau \in \{wl, wu, sl, su\}$) are subsets of the zeros of the terms in ψ_1 and ψ_2, respectively. By Lemma 3.11, $S^x_{\bar a}(\psi) = S^x_{\bar a}(\psi_1) \cap S^x_{\bar a}(\psi_2)$ is also a union of finitely many disjoint maximal intervals and $B_\tau\big(S^x_{\bar a}(\psi)\big) \subseteq B_\tau\big(S^x_{\bar a}(\psi_1)\big) \cup B_\tau\big(S^x_{\bar a}(\psi_2)\big)$ for $\tau \in \{wl, wu, sl, su\}$. Therefore, all finite bounds in $S^x_{\bar a}(\psi)$ are zeros of terms in ψ.

$\psi = \psi_1 \vee \psi_2$: By the semantics of $\vee$ and the definition of $S^x_{\bar a}$ we have $S^x_{\bar a}(\psi_1 \vee \psi_2) = S^x_{\bar a}(\psi_1) \cup S^x_{\bar a}(\psi_2)$. The same reasoning as for $\psi_1 \wedge \psi_2$ applies.

interval type	test point
$]-\infty,c]$	c
$[c,d[$	c
$[c,d]$	c
$]d,c]$	c
$[c,\infty[$	c
$]-\infty,c[$	$c-1$
$]c,d[$	$\frac{1}{2}(c+d)$
$]c,\infty[$	$c+1$
$]-\infty,\infty[$	0

Table 3.1: Interval types and suitable test points for them

$\psi \in At(\mathcal{V},\Sigma_{ord})$: ψ is of the form $t \cdot x + t' \,\rho\, 0$ with $\rho \in \{<,\leq,=,\neq,\geq,>\}$. Let $a = t^{\mathbb{R}}(\bar{a})$, $b = t'^{\mathbb{R}}(\bar{a})$, and $c = -\frac{b}{a}$ if $a \neq 0$. The satisfaction set $S^x_{\bar{a}}(\psi)$ depends on the values of a and b as is shown in the following table:

$S^x_{\bar{a}}(\psi)$	$a<0$	$a=0 \wedge b<0$	$a=0 \wedge b=0$	$a=0 \wedge b>0$	$a>0$
$<$	$]c,\infty[$	$\mathbb{R}$	$\varnothing$	$\varnothing$	$]-\infty,c[$
$\leq$	$[c,\infty[$	$\mathbb{R}$	$\mathbb{R}$	$\varnothing$	$]-\infty,c]$
$=$	$[c,c]$	$\varnothing$	$\mathbb{R}$	$\varnothing$	$[c,c]$
$\neq$	$\mathbb{R}\setminus\{c\}$	$\mathbb{R}$	$\varnothing$	$\mathbb{R}$	$\mathbb{R}\setminus\{c\}$
$\geq$	$]-\infty,c]$	$\varnothing$	$\mathbb{R}$	$\mathbb{R}$	$[c,\infty[$
$>$	$]-\infty,c[$	$\varnothing$	$\varnothing$	$\mathbb{R}$	$]c,\infty[$

Obviously, in every case the satisfaction set $S^x_{\bar{a}}(\psi)$ is a union of finitely many disjoint maximal intervals (note that $\mathbb{R}\setminus\{c\} =]-\infty,c[\cup]c,\infty[$) and the only finite bound appearing in the intervals is c, the solution of the equation $t \cdot x + t' = 0$. When $\rho \in \{<,\neq,>\}$, c is a strict bound, and when $\rho \in \{\leq,=,\geq\}$ then c is a weak bound.

□

The proof of Lemma 3.12 shows that all the finite interval bounds appearing in the satisfaction set of ψ are a subset of the zeros of the equations $t \cdot x + t' = 0$ where $t \cdot x + t' \,\rho\, 0$ with $\rho \in \{<,\leq,=,\neq,\geq,>\}$ is an atomic formula in ψ. Furthermore, atomic formulas with $\rho \in \{<,\neq,>\}$ contribute only to the strict bounds, and formulas with $\rho \in \{\leq,=,\geq\}$ contribute only to the weak bounds.

Finding Test Points We can now state how the quantifier elimination method really works. To eliminate the existential quantifier in $\exists x \psi$ the idea is to take a "test point" from every maximal interval of the satisfaction set. Since the satisfaction set is the set of all values for x where ψ becomes true, $\exists x \psi$ is true if and only if at least one of the chosen test points makes ψ true. Things are a bit complicated by the fact that we do not know the intervals themselves but the atomic formulas of ψ give us supersets of the bounds appearing in the intervals. Therefore, we have to choose a test point for every *possible* interval which has its bounds among the sets derived from the atomic formulas.

Table 3.1 shows which test points can be chosen for the different types of intervals in the satisfaction set. As one can see, if an interval has a weak bound, that weak bound can be chosen as test point (since the weak bound is part of the interval). Unfortunately the situation is more complex if the interval has strict bounds only. In the case of $]-\infty,c[$ and $]c,\infty[$ the point $c-1$ or $c+1$ is a valid choice for every (possible) interval. But for an interval of the form $]c,d[$ which has two strict bounds the values $c+1$ or $d-1$ cannot be used since $d-c$ could be less than 1. Therefore, every *pair of strict bounds* generates a new test point, namely $\frac{1}{2}(c+d)$. This gives rise to a quadratic amount of test points in the number of strict inequalities.

For the example $\alpha = \big((x \geq 0 \wedge 2x \leq p) \vee x > 7\big)$ we have the weak interval bounds 0, $\frac{p}{2}$, and the strict bound 7. According to Table 3.1 we have to choose as test points

- 0 for the *possible* intervals $]-\infty,0]$, $[0,\infty[$, $[0,7[$, $]7,0]$, $[0,\frac{p}{2}]$, $[\frac{p}{2},0]$, $]-\infty,\infty[$,
- $\frac{p}{2}$ for the *possible* intervals $]-\infty,\frac{p}{2}]$, $[\frac{p}{2},\infty[$, $[\frac{p}{2},0]$, $[0,\frac{p}{2}]$, $[\frac{p}{2},7[$, $]7,\frac{p}{2}]$,
- $6 = 7-1$ for the *possible* interval $]-\infty,7[$,
- $8 = 7+1$ for the *possible* interval $]7,\infty[$.

We see that superfluous test points can be chosen. By looking at the formula α, it is clear that 7 can only be a strict lower bound and there is no need to use 6 as a test point. In general, it is not possible to say whether an atomic formula gives rise to an upper or a lower bound (if it is the cause for a bound at all). This can be easily seen on the example $u_1 \cdot x + 1 > 0$. Therefore, we simply use every bound found in a formula ψ as a possible lower and a possible upper bound. Of course, the elimination set (cf. Definition 3.14) given in Theorem 3.16 can be optimized by the knowledge that some zeros of terms can only be lower or upper bounds. But, since we only show the principal idea of virtual substitution here, we do not go into possible optimizations.

Division Operations The above discussion of our selection of test points uses a division operation in two places. The (possible) interval bounds are solutions of equations $a \cdot x + b = 0$ if $a \neq 0$. The solution is then $-\frac{b}{a}$. For two strict bounds c and d it is necessary to use $\frac{1}{2}(c+d)$ as test point. We cannot substitute formulas with division symbols into the formula ψ, since no division symbol is part of the signature Σ_{ord}. Instead we try to find a Σ_{ord}-formula which is "equivalent" to the formula resulting from substituting a fraction into the given formula.

We extend the signature Σ_{ord} to the signature $\Sigma_{ord'}$ with the additional unary function symbol $\overline{{}^{-1}}$. We also extend the structure $\underline{\mathbb{R}}$ to the structure $\underline{\mathbb{R}}'$ by defining

$$a^{\overline{-1}} = \begin{cases} \frac{1}{a} & \text{if } a \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

When we substitute a quotient $s \cdot s'^{\overline{-1}}$ into an atomic formula α over Σ_{ord}, the result is a $\Sigma_{ord'}$-formula β. But it is always possible to find a Σ_{ord}-formula γ such that, for every environment $h \in \mathbb{R}^{\mathcal{V}}$, $\underline{\mathbb{R}}' \models_h (\beta \wedge s' \neq 0)$ if and only if $\underline{\mathbb{R}} \models_h (\gamma \wedge s' \neq 0)$. In other words, since we are working in the real numbers, the absence of a division symbol does not limit our expressivity.

As an example consider the formula $5 \cdot x + 3 > 0$ and let us substitute $\frac{a}{b}$ for x (assuming $b \neq 0$). The intermediate result is the $\Sigma_{ord'}$-formula $5 \cdot a \cdot b^{\overline{-1}} + 3 > 0$. Since b is not zero, b^2 is positive and we can multiply both sides of the inequality to yield (after canceling) $5 \cdot a \cdot b + 3 \cdot b^2 > 0$, which is a Σ_{ord}-formula. In case of an equation or an inequality with the relation $\neq$, it suffices to multiply with the denominator b (instead of b^2). The formalization of this idea is called *virtual substitution* and is presented in the next definition.

Definition 3.13 Let $t \cdot x + t' \,\rho\, 0$ be an atomic formula with $\rho \in \{<, \leq, =, \neq, \geq, >\}$. The *virtual substitution* of x by $s \cdot \overline{s'^{-1}}$ ($s, s' \in Tm(\mathcal{V}, \Sigma_{ord})$) is defined as

$$(t \cdot x + t' \,\rho\, 0)[x /\!\!/ s \cdot \overline{s'^{-1}}] := \begin{cases} t \cdot s \cdot s' + t' \cdot s'^2 \,\rho\, 0 & \text{if } \rho \in \{<, \leq, \geq, >\} \\ t \cdot s + t' \cdot s' \,\rho\, 0 & \text{if } \rho \in \{=, \neq\} \end{cases}$$

For any term $s \in Tm(\mathcal{V}, \Sigma_{ord})$ we define

$$(t \cdot x + t' \,\rho\, 0)[x /\!\!/ s] := (t \cdot x + t' \,\rho\, 0)[x/s]$$

This definition of virtual substitution can be extended to arbitrary first-order formulas in the same way as usual substitution.

Virtual substitution avoids division operations in formulas after substituting a fraction $s \cdot \overline{s'^{-1}}$ for a variable x by multiplying every equation and inequality in the intermediate $\Sigma_{ord'}$-formula by the (square of) the denominator of the fraction substituted for the variable. There is one important thing we have to take care of when we perform the virtual substitution $\psi[x /\!\!/ s \cdot \overline{s'^{-1}}]$. The formula $\psi[x /\!\!/ s \cdot \overline{s'^{-1}}]$ does not ensure that the denominator s' is not zero. This is consistent with the definition of $\overline{^{-1}}$, which is a totalization of the usual reciprocation operation $^{-1}$, i.e., it is also defined for the value 0. We must not forget that we have to make a case distinction on whether the denominator is zero or not. This is formalized in the following definition of elimination sets, where each test point t is guarded by a condition γ which ensures that the denominators are not zero. Elimination sets are a central concept of the quantifier elimination procedure shown in this section, and we present an example for an elimination set and the use of virtual substitution after the definition.

Definition 3.14 An *elimination set* for $\exists x \psi$ is a finite set $E \subseteq Qf(\mathcal{V}, \Sigma_{ord}) \times Tm(\mathcal{V}, \Sigma_{ord'})$ with the property that

$$\underline{\mathbb{R}} \models \exists x \psi \leftrightarrow \bigvee_{(\gamma,t) \in E} (\gamma \wedge \psi[x /\!\!/ t])$$

and $frvar(\gamma) \cup var(t) \subseteq \{u_1, \ldots, u_m\}$ for all $(\gamma, t) \in E$.

As has been stated before, the idea behind the quantifier elimination procedure presented here is to substitute x by a finite number of selected test points, such that if $\exists x \psi$ holds, at least one of the test points makes ψ true. The elimination set E contains elements (γ, t) where t represents one of the chosen test points and γ is an additional condition for the applicability of the test point t. The necessity of additional conditions can be seen from the following example.

Consider the equation $\alpha = (a \cdot x - 1 = 0)$. $\exists x \alpha$ is equivalent to the quantifier-free formula $a \neq 0$. The test points for this equation are $t := 1 \cdot \overline{a^{-1}}$, if $a \neq 0$, and 0 independently of a (cf. Theorem 3.16 below). Note that $\alpha[x /\!\!/ t] = (a \cdot 1 - 1 \cdot a = 0)$, i.e., $\alpha[x /\!\!/ t]$ holds for *every* $a \in \mathbb{R}$, but obviously α is false for $a = 0$. Therefore $\alpha[x /\!\!/ t]$ has to be guarded by the prerequisite $a \neq 0$. In this case, a correct elimination set would be $E = \{(a \neq 0, 1 \cdot \overline{a^{-1}}), (0 = 0, 0)\}$ (which makes an appropriate case distinction for $a \neq 0$) and the quantifier-free equivalent to $\exists x \alpha$ is

$$(a \neq 0 \wedge a \cdot 1 - 1 \cdot a = 0) \;\vee\; (0 = 0 \wedge a \cdot 0 - 1 = 0)$$

which is equivalent to the expected $a \neq 0$.

If we compare Definition 3.14 to Definition 3.1 of quantifier elimination, we see that the formula $\eta := \bigvee_{(\gamma,t)\in E} (\gamma \wedge \psi[x /\!/ t])$ has (at most) the free variables $u_1,\dots,u_m$ since $frvar(\psi) \subseteq \{u_1,\dots,u_m,x\}$ has been assumed. In addition, η is quantifier-free. If we choose the elimination set E as required by the definition, η is in fact a quantifier-free formula equivalent to $\exists x\, \psi$.

The remaining problem is to find a correct elimination set. First we define the set of critical points $C^x_s(\psi)$ and $C^x_w(\psi)$ for the strict and weak bounds defined by the atomic formulas of ψ. For atomic formulas $a \cdot x + b\, \rho\, 0$ (with $\rho \in \{<,\leq,\neq,=,\geq,>\}$) in ψ, the points $\frac{-b}{a}$ (for $a \neq 0$) are said to be "critical" since they determine the test points for the formula $\exists x\, \psi$ (see also the proof of Lemma 3.12 and the discussion of the choice of test points following that lemma).

Definition 3.15 The sets of critical points $C^x_s(\psi)$ and $C^x_w(\psi)$ are defined by

$$C^x_s(\psi) := \left\{ (t \neq 0, -t' \cdot \overline{t^{-1}}) \mid (t \cdot x + t'\, \rho\, 0) \text{ atomic formula of } \psi, \rho \in \{<,\neq,>\} \right\}$$
$$C^x_w(\psi) := \left\{ (t \neq 0, -t' \cdot \overline{t^{-1}}) \mid (t \cdot x + t'\, \rho\, 0) \text{ atomic formula of } \psi, \rho \in \{\leq,=,\geq\} \right\}$$

Using the critical points, it is now possible to define an elimination set for the formula $\exists x\, \psi$. As has been suggested in the discussion after Lemma 3.12, for weak inequalities and equations the critical points are chosen as test points, whereas for every critical point c of the strict inequalities the points $c-1$ and $c+1$ and for every pair of critical points of strict inequalities the arithmetic mean $\frac{1}{2}(c+d)$ has to be chosen as test point. Finally, the point 0 is an additional test point in case that ψ holds for all x and the atomic formulas do not generate any critical points (e.g., $\exists x\,(1 > 0)$).

Theorem 3.16 *The following set is an elimination set for $\exists x\, \psi$:*

$$\begin{aligned}
&\{(\gamma,t) \mid (\gamma,t) \in C^x_w(\psi)\} \cup \\
&\{(\gamma,t-1) \mid (\gamma,t) \in C^x_s(\psi)\} \cup \\
&\{(\gamma,t+1) \mid (\gamma,t) \in C^x_s(\psi)\} \cup \\
&\{(\gamma_1 \wedge \gamma_2, (t_1+t_2) \cdot \overline{2^{-1}}) \mid (\gamma_1,t_1),(\gamma_2,t_2) \in C^x_s(\psi),\ (\gamma_1,t_1) \neq (\gamma_2,t_2)\} \cup \\
&\{(0=0,0)\}
\end{aligned}$$

We have to note that the expressions $t-1$, $t+1$, and $(t_1+t_2)\cdot\overline{2^{-1}}$ do not satisfy the requirements of virtual substitution (Definition 3.13) since, due to the definition of $C^x_s(\psi)$, t_1+t_2 is not an element of $Tm(\mathcal{V},\Sigma_{ord})$ but of $Tm(\mathcal{V},\Sigma_{ord'})$, for example. This technical difficulty can be easily solved by applying usual arithmetic for fractions to $(t_1+t_2)\cdot\overline{2^{-1}}$ to transform it into a term of the form $s_1 \cdot \overline{s_2^{-1}}$.

Putting all these results together, we see that the elimination algorithm for the formula $\exists x\, \psi$ can be performed in three steps:

(1) Calculate $C^x_s(\psi)$ and $C^x_w(\psi)$.

(2) Calculate the elimination set E.

(3) The equivalent formula is then $\bigvee_{(\gamma,t)\in E} (\gamma \wedge \psi[x /\!/ t])$.

Example To demonstrate the complete procedure we consider the formula

$$\varphi = \exists x\,(p \cdot x + q \geq 0 \wedge x > q)$$

and try to compute a quantifier-free equivalent. The matrix ψ of φ contains two atomic formulas, $p \cdot x + q \geq 0$ and $x > q$. The zero of $p \cdot x + q$ is $\frac{-q}{p}$ if $p \neq 0$, and the zero of $x - q$ is q. The critical points of ψ are therefore:

$$\begin{aligned} C_w^x(\psi) &= \{(p \neq 0, (-q) \cdot \overline{p^{-1}})\} \\ C_s^x(\psi) &= \{(1 \neq 0, q)\} \end{aligned}$$

An elimination set according to Theorem 3.16 is then:

$$E = \{(p \neq 0, (-q) \cdot \overline{p^{-1}}), (1 \neq 0, q-1), (1 \neq 0, q+1), (0 = 0, 0)\}$$

and φ is equivalent to:

$$\begin{aligned} & \bigvee_{(\gamma,t) \in E} (\gamma \wedge \psi[x /\!/ t]) \\ = \quad & \big(p \neq 0 \wedge p \cdot (-q) \cdot p + q \cdot p^2 \geq 0 \wedge (-q) \cdot p - q \cdot p^2 > 0\big) \\ & \vee \big(1 \neq 0 \wedge p \cdot (q-1) + q \geq 0 \wedge (q-1) - q > 0\big) \\ & \vee \big(1 \neq 0 \wedge p \cdot (q+1) + q \geq 0 \wedge (q+1) - q > 0\big) \\ & \vee \big(0 = 0 \wedge p \cdot 0 + q \geq 0 \wedge 0 - q > 0\big) \end{aligned}$$

This can be simplified to (the second and fourth disjunct contain a contradiction):

$$\big(p \neq 0 \wedge p \cdot q \cdot (p+1) < 0\big) \;\vee\; \big(p \cdot (q+1) + q \geq 0\big)$$

and this is a quantifier-free equivalent of φ in $\mathbb{R}$.

3.2.3 Virtual Substitution and Non-linear Terms

The previous section shows how the elimination works for a formula which is linear in the quantified variable. The principle stays the same for non-linear formulas, but the chosen test points become in general more complex, i.e., they can contain not only quotients but also roots and that makes the definition of the virtual substitution more complex.

If we only eliminate variables which occur linearly in a formula, the elimination method for linear formulas is sufficient. To be more precise: if $\psi \in Qf(\mathcal{V}, \Sigma)$ is linear in $X \subseteq \mathcal{V}$ and $x \in X$, then the quantifier-free equivalent of $\exists x\, \psi$ is linear in $X \setminus \{x\}$. A proof for this is contained in the proof of Lemma 3.17.

The quantifier elimination with answer method for liner formulas described in Section 3.2.5 and the decision method for arbitrary variable free formulas (based on cylindrical algebraic decomposition) described in Section 3.3 are sufficient to solve all the problems we present in Chapter 4.

Virtual substitution has been generalized to support non-linear quantified variables. We are not discussing this here; the interested reader is referred to [Wei97] and [Wei94].

3.2.4 Multiple Quantifiers

When a formula φ contains more than one quantifier, the elimination is performed one quantifier at a time, starting with the innermost quantifier. Let $\varphi = Q_1\,x_1\,\cdots\,Q_{n-1}\,x_{n-1}\,Q_n\,x_n\,(\psi)$ be a prenex formula with $Q_1,\ldots,Q_n \in \{\exists,\forall\}$ and ψ quantifier-free. The quantifiers are eliminated as follows:

- If $Q_n = \exists$ then a quantifier-free formula ψ' which is equivalent to $Q_n\,x_n\,(\psi)$ can be computed and the elimination continues recursively with $Q_1\,x_1\,\cdots\,Q_{n-1}\,x_{n-1}\,(\psi')$.

- If $Q_n = \forall$ then $\forall\,x_n\,\psi$ is equivalent to $\neg\exists\,x_n\,(\neg\psi)$ and a quantifier-free formula ψ' which is equivalent to $\exists\,x_n\,(\neg\psi)$ can be computed. The elimination then continues recursively with $Q_1\,x_1\,\cdots\,Q_{n-1}\,x_{n-1}\,(\neg\psi')$.

3.2.5 Quantifier Elimination with Answer

A quantifier elimination method (as in Definition 3.1) computes for a given formula φ an equivalent, quantifier-free formula φ'. Sometimes it is desirable not only to know that there are values for the existentially quantified variables that make φ true, but also to get examples of such values together with φ'. Virtual substitution can give such examples and this enhanced algorithm is called "quantifier elimination with answer" or "extended quantifier elimination."

The answers for the existentially quantified variables are simply the test points from the elimination set.

Let $\varphi = \exists x_1 \ldots \exists x_n\,(\psi)$ for an arbitrary formula ψ with extension $(u_1,\ldots,u_m)$. The result of a quantifier elimination with answer is a set of pairs $(\gamma_i, \{x_1 = t_{i,1},\ldots,x_n = t_{i,n}\})$, for $i \in \{1,\ldots,l\}$, where φ is equivalent to $\gamma_1 \vee \cdots \vee \gamma_l$ and if γ_i holds (under given values for the parameters $u_1,\ldots,u_m$) then the terms $t_{i,j} \in Tm(\{u_1,\ldots,u_m\},\Sigma_{ord'})$ $(j \in \{1,\ldots,n\})$ in the associated substitution list $\{x_1 = t_{i,1},\ldots,x_n = t_{i,n}\}$ represent answers for $x_1,\ldots,x_n$ which make φ true in dependence of the parameters $u_1,\ldots,u_m$.

Example $\quad \varphi = \exists x(u_1 \cdot x + u_2 = 0 \wedge x \geq 0)$
A possible result of extended quantifier elimination is

$$
\begin{aligned}
&(u_1 > 0 \wedge u_2 \leq 0,\ \{x = -\tfrac{u2}{u1}\})\\
&(u_1 < 0 \wedge u_2 \geq 0,\ \{x = -\tfrac{u2}{u1}\})\\
&(u_1 = 0 \wedge u_2 = 0,\ \{x = 42\})
\end{aligned}
$$

Note that the choice $x = 42$ in the case of $u_1 = 0 \wedge u_2 = 0$ is completely arbitrary, since any non-negative real number satisfies the formula in this case. Which representative of the solutions is returned by the extended quantifier algorithm depends on the implementation, so we cannot make assumptions about which solution is delivered (except, of course, if we depend on implementation details). If one desires a specific solution, one must add constraints to the formula which express the properties of the desired solution. For example, the formula $\exists x\,\big(\psi \wedge \forall y\,(\psi[x/y] \rightarrow x \leq y)\big)$ with $\psi = (u_1 \cdot x + u_2 = 0 \wedge x \geq 0)$ has (compared to φ) the additional constraint that the solution for x is minimal, so for this input the answer for the case $u_1 = 0 \wedge u_2 = 0$ must be $x = 0$.

Computing Answers in General To compute answers with virtual substitution, the quantifier elimination for a formula $\exists x_1 \dots \exists x_n \psi$ is performed as follows. First, the quantifiers in the formula ψ are eliminated to yield the equivalent formula ψ'. Then, as usual, the elimination continues with $\exists x_n \psi'$. The elimination set for this formula $E = \{(\gamma_1, t_1), \dots, (\gamma_k, t_k)\}$ contains conditions $\gamma^{(n)}$ and solutions $t^{(n)}$ for x_n. For every element $(\gamma^{(n)}, t^{(n)}) \in E$ one continues the elimination for the formula $\exists x_{n-1}(\gamma^{(n)} \wedge \psi'[x_n /\!\!/ t^{(n)}])$. This is repeated for the quantifiers $\exists x_{n-2}, \dots, \exists x_1$, which yields a set of results each of which is of the form

$$\gamma := \gamma^{(1)} \wedge \big(\dots \big(\gamma^{(n-1)} \wedge (\gamma^{(n)} \wedge \psi'[x_n /\!\!/ t^{(n)}])[x_{n-1} /\!\!/ t^{(n-1)}]\big) \dots\big)[x_1 /\!\!/ t^{(1)}]$$

Each of these results corresponds to one of the partial solutions $(\gamma, \{x_1 = t^{(1)}, \dots, x_n = t^{(n)}\})$ of the extended quantifier elimination.

The following lemma states that the linearity of the input formula ensures the linearity of the answers.

Lemma 3.17 *Let $\psi \in Tm(\mathcal{V}, \Sigma)$ be linear in the variables $\{x_1, \dots, x_n\}$ and let*

$$\big\{\big(\gamma_i, \{t_{i,1}, \dots, t_{i,n}\}\big) \quad | \quad i \in \{1, \dots, k\}\big\}$$

be the answer computed for the question $\exists x_1 \dots \exists x_n \psi$ (for some suitable $k \in \mathbb{N}$). Then every $t_{i,j}$ is linear in $\{x_1, \dots, x_{i-1}\}$ (and $x_i, \dots, x_n \notin var(t_{i,j})$ by the definition of quantifier elimination with answer).

Proof by induction on n.

$n = 1$: All the solutions $t_{1,j}$ for x_1 in the formula $\exists x_1 \psi$ are $\Sigma_{ord'}$-terms in the constants only. This implies trivially that the $t_{1,j}$ are linear in $\{x_1, \dots, x_0\} = \varnothing$ due to the Definition 3.8 of linear terms.

$n \to n+1$: Let $\varphi = \exists x_{n+1} \psi$. Since we assume that ψ is linear in $\{x_1, \dots, x_{n+1}\}$, every term in ψ can be written as $\sum_{l=1}^{n+1} a_l x_l + a_0$, where $a_0, \dots, a_{n+1}$ are appropriately chosen terms with $x_1, \dots, x_{n+1} \notin var(a_0) \cup \dots \cup var(a_{n+1})$. Therefore, the critical points of ψ are each of the form $\big(-a_0 - \sum_{l=1}^{n} a_l x_l\big) \cdot \overline{a_{n+1}^{-1}}$. The test points of the elimination set, as defined by Theorem 3.16, are derived form the critical points and each of them can again be written in the form $\big(-b_0 - \sum_{l=1}^{n} b_l x_l\big) \cdot \overline{b_{n+1}^{-1}}$ for suitable terms $b_0, \dots, b_{n+1}$ with $x_1, \dots, x_{n+1} \notin var(b_0), \dots, var(b_{n+1})$. Rewriting these terms in the form $-b_0 \cdot \overline{b_{n+1}^{-1}} - \sum_{l=1}^{n}(b_l \cdot \overline{b_{n+1}^{-1}} \cdot x_l)$ shows that the answers for x_{n+1} are linear in $\{x_1, \dots, x_n\}$. Definition 3.13 of virtual substitution ensures that $\psi[x /\!\!/ t]$, where t is such a test point, is linear in $\{x_1, \dots, x_n\}$. In addition, the conditions γ in the elimination set (cf. Definition 3.16) are linear in $\{x_1, \dots, x_n\}$. Therefore, the quantifier-free equivalent φ' of φ is linear in $\{x_1, \dots, x_n\}$. Applying the induction hypothesis to $\exists x_1 \dots \exists x_n \varphi'$ yields the remaining parts of the proposition.

□

3.2.6 Generalized Method with Infinitesimals

As has been note above, strict inequalities give rise to quadratically many test points in the elimination set, since every pair of strict inequalities could cause the satisfaction set to contain a maximal interval of the form $]c, d[$ and the choice $\frac{1}{2}(c+d)$ of the test point depends on both c and d. To reduce the number of test points, a variant of virtual substitution has been developed which allows to introduce infinitesimally small quantities ("ε") and unbounded large quantities ("∞") during the elimination. The virtual substitution has to be extended to deal with epsilons and infinities, but then the number of test points can be reduced considerably.

A possible elimination set for $\exists x\,(\psi)$ is then

$$\begin{aligned}&\{(\gamma,t) \mid (\gamma,t) \in \mathcal{C}_w^x(\psi)\} \cup\\&\{(\gamma,t+\varepsilon) \mid (\gamma,t) \in \mathcal{C}_s^x(\psi)\} \cup\\&\{(0=0,-\infty)\}\end{aligned}$$

More on this can be found in [LW93].

Unfortunately, this approach has a drawback. If one performs quantifier elimination with answer, the epsilons and infinities can appear in the answers for the existentially quantified variables. This can make the result hard to interpret. In general, the substitution $\varepsilon = 0$ invalidates the result, since an ε is always introduced with the constraint $\varepsilon > 0$. On the other hand, nothing is known about the magnitude of ε and it is not possible to choose some (arbitrary) small positive value for ε.

The implementation of virtual substitution we use (REDLOG [DS97a]) uses this generalized method with infinitesimals. Therefore, we have to take care when we ask it to solve a quantifier elimination with answer problem. The applications of quantifier elimination with answer we present in Chapter 4 are formulated such that the answers do not contain infinitesimals. Unfortunately, this makes some of the problems more complex (e.g., Section 4.3.2), but there is currently no implementation of virtual substitution available which does not use infinitesimals.

3.3 Cylindrical Algebraic Decomposition

The idea behind quantifier elimination with virtual substitution is to find a test point for every possible maximal interval of the satisfaction set of the matrix of an existentially quantified formula. Cylindrical algebraic decomposition (CAD) also calculates test points which are substituted into the matrix of a formula. But, in contrast to virtual substitution, CAD does not take the satisfaction set of the matrix into account. Instead, they are based on the following consideration.

The terms of a Σ_{ord}-formula ψ are polynomials $p_1,\ldots,p_n$. If we choose test points $t_1,\ldots,t_k$ which cover every possible sign combination of the polynomials $p_1,\ldots,p_n$, we can test the formulas $\exists(\psi)$ and $\forall(\psi)$ by verifying that at least one or, respectively, every test point satisfies ψ.

Cylindrical algebraic decomposition does not compute a minimal set of test points to cover every possible sign combination of the given polynomials. It uses a projection method to project the polynomials to have lower dimensionality until univariate polynomial are reached. A cylindrical decomposition for the 1-dimensional case is then computed using root isolation. Finally, the decomposition is extended to higher dimensionalities until the dimensionality of the original polynomials is reached. Our presentation of CAD is based on [ACM98] and [Hon98].

First we first give some definitions we need to introduce cylindrical algebraic decomposition.

3.3.1 Definitions

Definition 3.18 A non-empty connected subset of $\mathbb{R}^r$ ($r \in \mathbb{N}$) is called a *region*. For a region R we define the *cylinder over R*, written as $Z(R)$, as $R \times \mathbb{R}$.

The cylinder over $\mathbb{R}^0 = \{()\}$ is $\mathbb{R}^0 \times \mathbb{R} = \mathbb{R}$.

Definition 3.19 Let R be a region of $\mathbb{R}^r$. An *f-section of* $Z(R)$ is the set

$$\Big\{\big(a, f(a)\big) \;\big|\; a \in R\Big\}$$

for a continuous function $f : R \to \mathbb{R}$.
An (f_1, f_2)*-sector of* $Z(R)$ is the set

$$\Big\{\big(a, b\big) \;\big|\; a \in R, b \in \mathbb{R}, f_1(a) < b < f_2(a)\Big\}$$

where $f_1 = -\infty$ or $f_1 : R \to \mathbb{R}$ is continuous, and $f_2 = \infty$ or $f_2 : R \to \mathbb{R}$ is continuous, and $f_1(x) < f_2(x)$ for every $x \in R$.

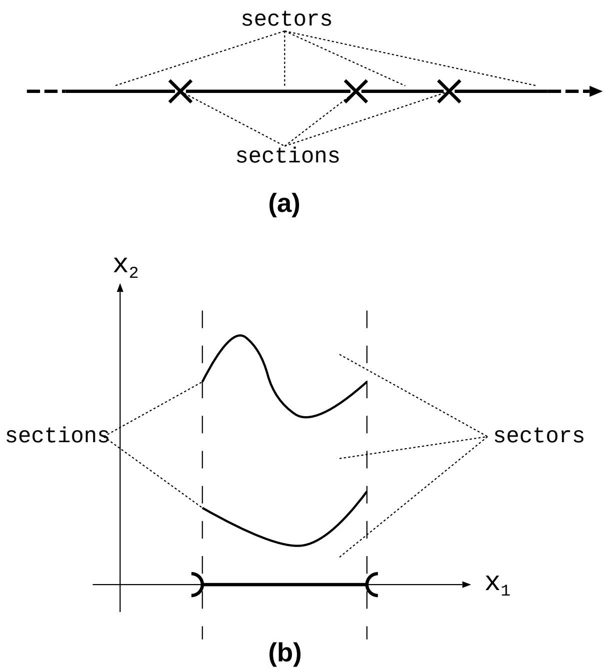


Figure 3.1: Sections and sectors

Obviously, sections and sectors are regions. Figure 3.1 shows some sections and sectors of $\mathbb{R}^1$ in (a), and some sections and sectors of a cylinder over an interval in $\mathbb{R}^2$ in (b). The sections and sectors shown in Figure 3.1 also form stacks as defined by the following definition.

Definition 3.20 Let $X \subseteq \mathbb{R}^r$. A *decomposition* of X is a finite collection of pairwise disjoint regions whose union is X.
Let R be a region, $k \in \mathbb{N}$, and $f_1, \ldots, f_k : R \to \mathbb{R}$ be continuous functions with $f_1(x) < f_2(x) < \cdots < f_k(x)$ for every $x \in R$. Then $(f_1, \ldots, f_k)$ defines a decomposition of $Z(R)$ consisting of the sets

- f_i-sections of $Z(R)$ for $1 \leq i \leq k$,
- (f_i, f_{i+1})-sectors of $Z(R)$ for $1 \leq i < k$,
- the $(-\infty, f_1)$-sector of $Z(R)$,
- the (f_k, ∞)-sector of $Z(R)$.

Such a decomposition is called a *stack over R* defined by $(f_1, \ldots, f_k)$.
In the case of $k = 0$ the decomposition consists only of the $(-\infty, \infty)$-sector of $Z(R)$, i.e., the stack consists of the single region $Z(R)$.

Definition 3.21 A decomposition D of $\mathbb{R}^r$ is called *cylindrical*, if either

(1) $r = 1$ and D is a stack over $\mathbb{R}^0$, or

(2) $r > 1$ and there is a cylindrical decomposition D' of $\mathbb{R}^{r-1}$ such that, for each region R of D', D contains a stack over R.

A decomposition D of $\mathbb{R}^r$ is called *algebraic* if each of its regions R is a semi-algebraic set, i.e., if R can be defined by a quantifier-free Σ_{ord}-formula.
A *cylindrical algebraic decomposition* (CAD) is a decomposition which is both cylindrical and algebraic.

Since the terms of Σ_{ord}-formulas are polynomials, we get a CAD if we construct a cylindrical decomposition whose stacks (i.e., sectors and sections) are defined by polynomials.

Definition 3.22 Let $X \subseteq \mathbb{R}^r$ and $p \in \mathbb{Q}[x_1, \ldots, x_r]$. We say *p is invariant on X* if one of the following conditions holds:

- $p(\alpha) > 0$ for all $\alpha \in X$,
- $p(\alpha) = 0$ for all $\alpha \in X$,
- $p(\alpha) < 0$ for all $\alpha \in X$.

We say that a set of polynomials $A \subset \mathbb{Q}[x_1, \ldots, x_r]$ is invariant on X, if every $p \in A$ is invariant on X. A decomposition D is called *A-invariant* if A is invariant on every region of D.

Our aim is to construct—for a given quantifier-free formula ψ—a cylindrical decomposition such that the terms of ψ are invariant on the regions of the decomposition. We are not interested in the regions of the decomposition themselves, but in computing a test point for each of the regions. Since the terms of ψ are invariant on each of the regions, the truth value of ψ is also invariant on the regions. We can then substitute the test points into the formula ψ to compute the truth value of ψ on each region. If ψ is true on every region, the formula $\forall(\psi)$ is true; if ψ is true on at least one region, the formula $\exists(\psi)$ is true.

3.3.2 Projection Phase

The projection phase of the CAD algorithm has the purpose to project a finite set of polynomials $A \subset \mathbb{Q}[x_1, \dots, x_r]$ to a finite set $\mathrm{PROJ}(A) \subset \mathbb{Q}[x_1, \dots, x_{r-1}]$ of polynomials in one indeterminate less. The condition this projection has to satisfy is: if D' is a CAD of $\mathbb{R}^{r-1}$ such that D' is $\mathrm{PROJ}(A)$-invariant, then it is possible to construct a stack over each region of D' such that all these stacks together form a CAD D which is A-invariant.

Central for the correctness of a projection is the notion of delineability.

Definition 3.23 Let $p \in \mathbb{Q}[x_1, \dots, x_r]$ and R be a region of $\mathbb{R}^{r-1}$. p is called *delineable* on R if the portion of the set of zeros $V(p) = \{\bar{x} \in \mathbb{R}^r \mid p(\bar{x}) = 0\}$ of p which lies in $Z(R)$ consists of pairwise disjoint sections of $Z(R)$.

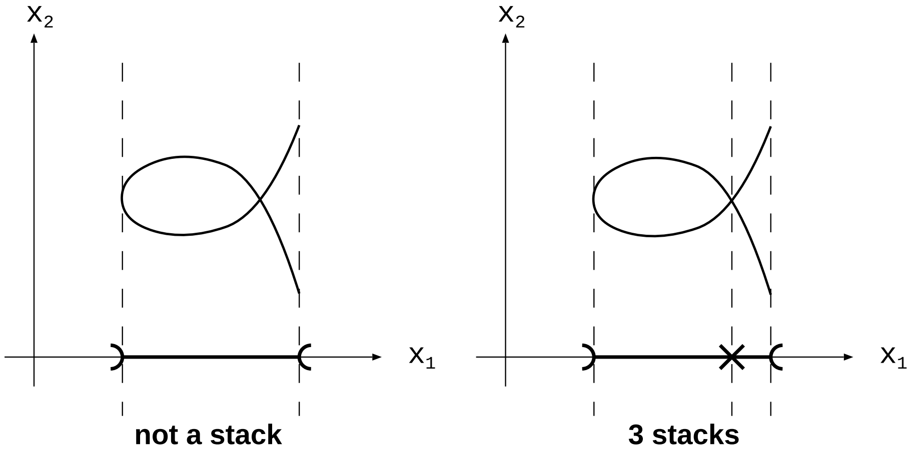


Figure 3.2: Delineability of a polynomial

To put this definition in other words: the zeros of p define a stack over R, since the different branches of the zeros of p "do not cross" over R. Figure 3.2 shows an interval and the zeros of a hypothetical polynomial p over that interval. p is not delineable over the whole interval and, therefore, the zeros of p do not define a stack over the interval, but p is delineable over the three regions outlined in the right part of the figure.

The correctness of a projection is characterized by two conditions. For any $\mathrm{PROJ}(A)$-invariant region R the following must hold:

(1) Each $p \in A$ is delineable on R, or $p(\bar{x}) = 0$ for every $\bar{x} \in Z(R)$.

(2) The sections of $Z(R)$ belonging to different $p, q \in A$ are either disjoint or identical.

Condition (1) ensures that every $p \in A$ gives rise to a stack over R (or p is the zero polynomial over R) and condition (2) states that all the polynomials in A together define a stack.

Different projection operations have been proposed in the literature. Our implementation is based on the projection operator from [Hon98]:

Theorem 3.24 *A projection operator satisfying the conditions (1) and (2) is*

$$\mathrm{PROJ}(A) := \bigcup_{\substack{p\in A\\ r\in\mathrm{RED}(p)}} \big(\{\mathrm{lc}(r)\}\cup\mathrm{PSC}(r,r')\big) \cup \bigcup_{\substack{p,q\in A\\ p<q}} \bigcup_{r\in\mathrm{RED}(p)} \mathrm{PSC}(r,q)$$

where $<$ denotes an arbitrary linear ordering of the polynomials in A.

For the operations RED, lc, and PSC, we view the polynomials $p \in \mathbb{Q}[x_1,\dots,x_r]$ as elements of $\big(\mathbb{Q}[x_1,\dots,x_{r-1}]\big)[x_r]$, i.e., we treat them as polynomials in x_r having coefficients from $\mathbb{Q}[x_1,\dots,x_{r-1}]$.

3.3.3 Base Case

The base case of the CAD algorithm is reached when—by applying the PROJ operator $r-1$ times—the polynomials in $A \subset \mathbb{Q}[x_1,\dots,x_r]$ have been projected to univariate polynomials in $\mathrm{PROJ}^{r-1}(A) \subset \mathbb{Q}[x_1]$. In the univariate case it is possible to compute a cylindrical algebraic decomposition by root isolation. The CAD is constructed as follows:

- First we make sure that no two polynomials have the same zero. To achieve this, we repeatedly take two polynomials $p,q \in A$ with $g := \gcd(p,q) \neq 1$ and replace the set A by the set $(A \setminus \{p,q\}) \cup \{\frac{p}{g}, \frac{q}{g}, g\}$ until the polynomials in A are pairwise relatively prime.
- Using root isolation, one can find isolation intervals for the zeros of every polynomial $p \in A$.
- The isolation intervals can be refined to be pairwise disjoint. The zeros $\rho_1 < \rho_2 < \cdots < \rho_k$ then define algebraic numbers

Algebraic number	Isolation interval	Defining polynomial
ρ_1	$[a_1,b_1]$	p_1
$\vdots$	$\vdots$	$\vdots$
ρ_k	$[a_k,b_k]$	p_k

- The regions of the CAD are
  $]-\infty,\rho_1[$, $\{\rho_1\}$, $]\rho_1,\rho_2[$, $\{\rho_2\}$, $\dots$, $]\rho_{k-1},\rho_k[$, $\{\rho_k\}$, $]\rho_k,\infty[$.
- Test points for the regions can be chosen according to the following table:

Region	Test point
$]-\infty,\rho_1[$	a_1
$\{\rho_1\}$	ρ_1
$]\rho_1,\rho_2[$	b_1
$\{\rho_2\}$	ρ_2
$\vdots$	$\vdots$
$]\rho_{k-1},\rho_k[$	b_{k-1}
$\{\rho_k\}$	ρ_k
$]\rho_k,\infty[$	b_k

- If the polynomials in A do not have any zeros, we choose $]-\infty,\infty[$ as the only region of the CAD with the test point 0.

Since the polynomials in A are continuous functions and the regions of the CAD are defined using the zeros of these polynomials, it is clear that the CAD is A-invariant. It is also clear from the above table of test points that we can choose rational numbers as test points for the regions which are intervals. The regions which consist only of a single zero have, of course, that zero as test point, and in general this is an algebraic number.

3.3.4 Complete CAD Procedure

After the projection phase and the base case, the extension phase constructs a CAD of $\mathbb{R}^r$, that is, test points for its regions, from given test points of a CAD of $\mathbb{R}^{r-1}$.

We show the extension phase as part of the complete CAD procedure which computes, given $A \subset \mathbb{Q}[x_1,\ldots,x_r]$, a set of test points T for the regions of an A-invariant CAD:

(1) If $A \subset \mathbb{Q}[x_1]$, we are in the base case and T is computed as outlined in Section 3.3.3.

(2) Otherwise:

  (a) Compute a set T' of test points for a PROJ(A)-invariant CAD of $\mathbb{Q}[x_1,\ldots,x_{r-1}]$ by recursively applying the CAD method to PROJ$(A) \subset \mathbb{Q}[x_1,\ldots,x_{r-1}]$.

  (b) For every $\alpha = (\alpha_1,\ldots,\alpha_{r-1}) \in T'$ compute

$$P_\alpha := \left\{ p(\alpha_1,\ldots,\alpha_{r-1},x_r) \;\middle|\; p \in A,\, p(\alpha_1,\ldots,\alpha_{r-1},x_r) \neq 0 \right\}$$

  (c) Isolate the zeros of the polynomials in P_α and construct test points $T_\alpha := \{\tau_{\alpha,1},\ldots,\tau_{\alpha,k_\alpha}\}$ from the zeros the same way it is done in the base case (Section 3.3.3) for every $\alpha \in T'$.

  (d) The test points representing the regions of an A-invariant CAD are

$$T := \{(\alpha_1,\ldots,\alpha_{r-1},\tau) \mid (\alpha_1,\ldots,\alpha_{r-1}) \in T', \tau \in T_{(\alpha_1,\ldots,\alpha_{r-1})}\}$$

Remarks:

- By the notation $p(\alpha_1,\ldots,\alpha_{r-1},x_r)$ we mean that we substitute $\alpha_1,\ldots,\alpha_{r-1} \in \mathbb{A}$ for the indeterminates $x_1,\ldots,x_{r-1}$ in p and obtain a univariate polynomial in the only indeterminate x_r.

- $p(\alpha_1,\ldots,\alpha_{r-1},x_r) \neq 0$ is used to express that, after substituting $\alpha_1,\ldots,\alpha_{r-1}$ into p, the resulting univariate polynomial is not the zero polynomial.

- The polynomials $p(\alpha_1,\ldots,\alpha_{r-1},x_r)$ in P_α are elements of $\mathbb{A}[x_r]$, i.e., the coefficients of these polynomials are algebraic numbers, in general. We can still apply the techniques of Section 2.3.4 to isolate the zeros of a $p(\alpha_1,\ldots,\alpha_{r-1},x_r)$, but a naive application of these techniques is extremely inefficient: all the computations to isolate the zeros of such a polynomial have to be performed with polynomials from $\mathbb{A}[x_r]$ and, therefore, the arithmetic operations on the coefficients of these polynomials are operations with algebraic numbers. There are different ways around these "stacked algebraic numbers":

– Using the algorithm NORMAL, one can compute a polynomial $q \in \mathbb{Q}[x_r]$ for every $p \in P_\alpha$ such that every zero of p is also a zero of q. The disadvantage of using q instead of p is that q has, in general, more zeros than p, and this causes more test points to be calculated than necessary.

– A more rigorous solution to this problem involves a more complex representation of algebraic numbers. The idea is to represent a test point $(\beta_1,\ldots,\beta_{r-1})$ by a primitive algebraic number γ and polynomials $b_1,\ldots,b_{r-1} \in \mathbb{Q}[x]$ such that $\beta_i = b_i(\gamma)$ (computed using the algorithm SIMPLE). The arithmetic with $\beta_1,\ldots,\beta_{r-1}$ can then be done using $b_1,\ldots,b_{r-1}$ in the quotient field $\mathbb{Q}[x]/(\mu)$ where $\mu \in \mathbb{Q}[x]$ is the minimal polynomial of γ.

3.3.5 Example

As an example for the computation of a cylindrical algebraic decomposition, let us look at the formula $\psi = (a-3 \geq 0 \rightarrow a^2+b^2-1 > 0)$ and the decision problem $\forall a \forall b\,(\psi)$. The formula ψ contains the two polynomials $p_1(a,b) = a^2+b^2-1$ and $p_2(a,b) = a-3$. The zeros of p_1 and p_2 are shown in Figure 3.3. The polynomial p_1 has a negative sign inside

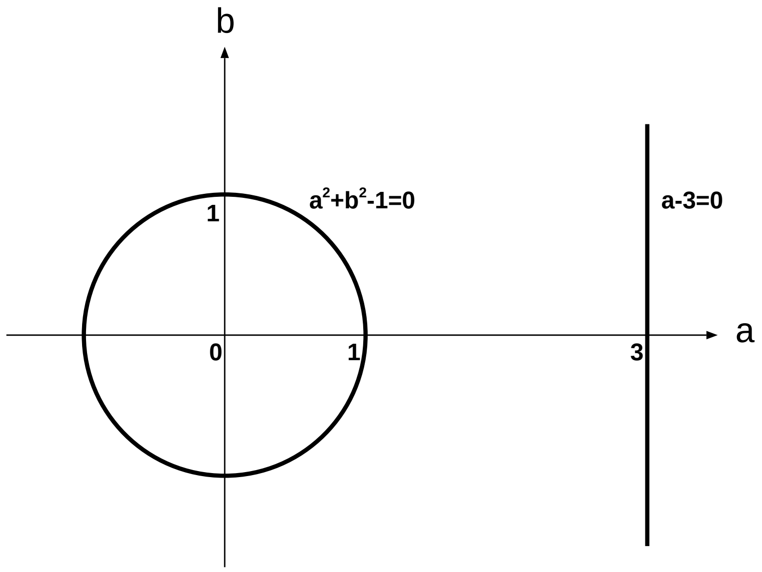


Figure 3.3: Zeros of the polynomials in the CAD example

the circle around the origin and a positive sign outside the circle; p_2 is negative left of the straight line and positive on the other side.

We are looking for a cylindrical algebraic decomposition of $\mathbb{R}^2$ which is sign invariant for p_1 and p_2. We choose to process the dimensions in the order b,a, i.e., we first project the b-dimension away, then construct a decomposition of the a-dimension, and finally extend the decomposition to $\mathbb{R}^2$.

By looking at the zeros of p_1 and p_2, we can see that, for $a < 1$, $1 < a < 3$, and $a > 3$, both polynomials have no zeros. At $a = -1$ and $a = 1$ the polynomial p_1 has one zero, and for $-1 < a < 1$ the zeros of p_1 consist of two separate branches. p_2 is identical to the zero polynomial for $a = 3$.

Projection Phase To calculate the projection of $\{p_1, p_2\}$ to the a-dimension, we need the reducta of p_1 and p_2 with respect to the variable b:

$$\begin{aligned}\mathrm{RED}(p_1) &= \{b^2 + (a^2-1), a^2-1\}\\ \mathrm{RED}(p_2) &= \{a-3\}\end{aligned}$$

We now compute $\mathrm{PROJ}(\{p_1, p_2\})$ (cf. Theorem 3.24). Note that the operations lc and res_k are performed with respect to b, which is the main variable of the input polynomials. First we show the calculation of the leading coefficients and the PSC sets of every reductum with its formal derivative:

$r = b^2 + (a^2-1)$:

$$\begin{aligned}&r' = 2b\\ &\mathrm{lc}(r) = 1\\ &\mathrm{res}_0(r, r') = \det\begin{pmatrix}1 & 2 & 0\\ 0 & 0 & 2\\ a^2-1 & 0 & 0\end{pmatrix} = 4a^2-4\\ &\mathrm{res}_1(r, r') = det\begin{pmatrix}2\end{pmatrix} = 2\\ &\mathrm{PSC}(r, r') = \{4a^2-4, 2\}\end{aligned}$$

$r = a^2 - 1$:

$$\begin{aligned}&r' = 0\\ &\mathrm{lc}(r) = a^2-1\\ &\mathrm{PSC}(r, r') = \varnothing\end{aligned}$$

$r = a - 3$:

$$\begin{aligned}&r' = 0\\ &\mathrm{lc}(r) = a-3\\ &\mathrm{PSC}(r, r') = \varnothing\end{aligned}$$

The second part of $\mathrm{PROJ}(\{p_1, p_2\})$ calculates PSC sets of reducta derived from different polynomials in $\{p_1, p_2\}$:

$r = b^2 + (a^2-1)$, $s = a-3$:

$$\begin{aligned}&\mathrm{res}_0(r, s) = \det\begin{pmatrix}a-3 & 0\\ 0 & a-3\end{pmatrix} = (a-3)^2\\ &\mathrm{PSC}(r, s) = \left\{(a-3)^2\right\}\end{aligned}$$

$r = a^2 - 1$, $s = a-3$:

$$\begin{aligned}&\mathrm{res}_0(r, s) = 1\\ &\mathrm{PSC}(r, s) = \{1\}\end{aligned}$$

Putting these results together, we have

$$\mathrm{PROJ}(\{p_1, p_2\}) = \left\{4a^2-4, a^2-1, (a-3)^2, 2, 1\right\}$$

Since 1 and 2 are constant polynomials and $4a^2-4$ is a multiple of a^2-1, we can use the set $A = \{a^2-1, (a-3)^2\}$ as input for the CAD computation in the a-dimension.

Base case With $A = \{a^2 - 1, (a-3)^2\}$ we have reached the base case, since the polynomials in A are univariate. The zeros of the polynomials in A are $\{-1, 1, 3\}$, and from this we easily construct the regions and test points of a CAD of the a-dimension:

region	test point
$]-\infty, -1[$	-2
$\{-1\}$	-1
$]-1, 1[$	0
$\{1\}$	1
$]1, 3[$	2
$\{3\}$	3
$]3, \infty[$	4

The regions constructed from the zeros of the polynomials in A are exactly the intervals we discovered by looking at the branches of the zeros of $\{p_1, p_2\}$.

Extension Phase The final step in our CAD computation is to extend the CAD of the a-dimension found in the base case to a $\{p_1, p_2\}$-invariant CAD of $\mathbb{R}^2$. For every region in the CAD of $\mathbb{R}^1$ already computed, we substitute the region's test point into the polynomials in $P = \{p_1, p_2\}$ (for the indeterminate a) and isolate the zeros of the resulting polynomials. We use the notation $P[a/t]$ as abbreviation for $\{p_1[a/t], p_2[a/t]\}$.

- region $]-\infty, -1[$:

  $P[a/-2] = \{b^2 + 3, -5\}$ No zeros, test point 0.

region	test point
$]-\infty, -1[\times\mathbb{R}$	$(-2, 0)$

- region $\{-1\}$:

  $P[a/-1] = \{b^2, -4\}$ One zero at $b = 0$, test points -1, 0, 1.

region	test point
$\{-1\}\times]-\infty, 0[$	$(-1, -1)$
$\{(-1, 0)\}$	$(-1, 0)$
$\{-1\}\times]0, \infty[$	$(-1, 1)$

- region $]-1, 1[$:

  $P[a/0] = \{b^2 - 1, -3\}$ Two zeros $b = -1$ and $b = 1$, test points -2, -1, 0, 1, 2.

region	test point
$\{(a, b) \mid -1 < a < 1, b < -a^2\}$	$(0, -2)$
$\{(a, -a^2) \mid -1 < a < 1\}$	$(0, -1)$
$\{(a, b) \mid -1 < a < 1, -a^2 < b < a^2\}$	$(0, 0)$
$\{(a, a^2) \mid -1 < a < 1\}$	$(0, 1)$
$\{(a, b) \mid -1 < a < 1, a^2 < b\}$	$(0, 2)$

- region $\{1\}$: Similar to region $\{-1\}$.
- region $]1, 3[$: Similar to region $]-\infty, -1[$.

- region $\{3\}$:
  $P[a/3] = \{b^2+8, 0\}$ The zero polynomial in $P[a/3]$ is ignored; b^2+8 has no zeros. The only test point is 0.

region	test point
$\{3\}\times\mathbb{R}$	$(3,0)$

- region $]3,\infty[$: Similar to region $]-\infty,-1[$.

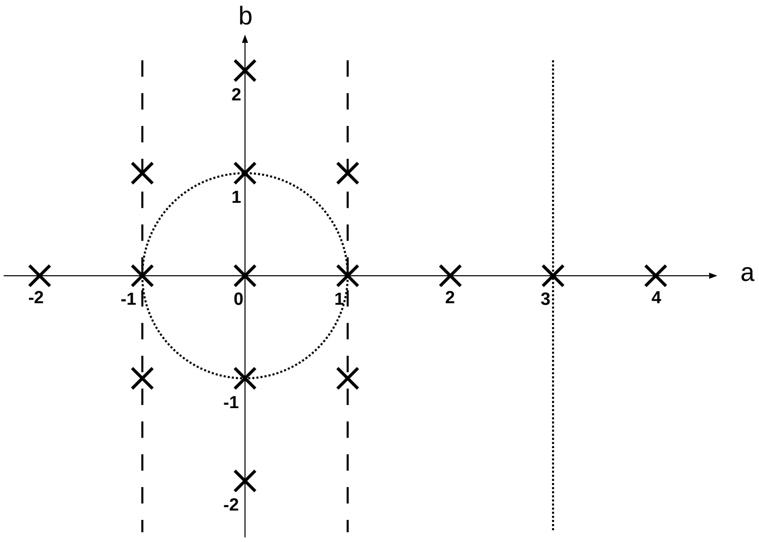


Figure 3.4: Test points for a $\{a^2+b^2-1, a-3\}$-invariant CAD

Figure 3.4 shows the test points for each region. These test points can be used to decide the formula $\forall a \forall b\, (a-3 \geq 0 \rightarrow a^2+b^2-1 > 0)$. If we evaluate $\psi = (a-3 \geq 0 \rightarrow a^2+b^2-1 > 0)$ for each of our test points, we see that ψ holds for every test point, and hence, $\forall a \forall b\, \psi$ holds in $\underline{\mathbb{R}}$.

3.4 Integers

Virtual substitution and cylindrical algebraic decomposition both work in the real numbers, and their answer for a question $\exists(\psi)$ states whether values for the variables in $frvar(\psi)$ exist such that ψ becomes true in the reals. They cannot be used to analyze the feasibility of a formula in the integers, in general. We are not elaborating much on integral solutions in this thesis since, in general, the existence of integral solutions is undecidable, and it turned out that quantifier elimination in the reals is sufficient to solve the problems we show in Chapter 5.

Here is the idea for finding the integral zeros of univariate polynomials. To decide whether integral zeros of a polynomial p exist, one can isolate the zeros of p and make every isolation interval $[a,b]$ so small that $b-a < 1$. The interval then contains at most one integral point. $[a,b]$ contains the integral point $\lceil a \rceil$ if and only if $\lceil a \rceil \leq b$. If $\lceil a \rceil$ is part of the interval, one checks whether $p(\lceil a \rceil) = 0$. With this method one can find every integral zero of p.

To find the integral solutions for $p > 0$, one constructs test points like in the CAD base case (Section 3.3.3) and tests $p(t) > 0$ for every such test point. From this, one infers which integral points between the zeros (and possibly below the smallest and above the biggest zero) solve $p > 0$. The procedure is analogous for $<, \leq, \geq, \neq$.

3.5 Implementations Used for this Thesis

The previous sections outline virtual substitution as a quantifier elimination procedure and cylindrical algebraic decomposition as a decision method. We use both algorithms in our practical experiments (Chapter 5):

- The commercial computer algebra system REDUCE [Hea99, Red] includes a quantifier elimination package (REDLOG [DS97a]) which is written by Andreas Dolzmann and Thomas Sturm and is based on virtual substitution.
- An implementation of cylindrical algebraic decomposition (and its sub-algorithms like root isolation, algebraic number arithmetic, SIMPLE, NORMAL) are provided together with other algorithms (e.g., the generalized Fourier-Motzkin method from Section 4.2.1) in our implementation. Although arithmetic in $\mathbb{Q}[x]/(\mu)$ (for a minimal polynomial μ) is available in the implementation,[1] the CAD procedure uses the "simple" method of replacing every $p \in P_\alpha \subset \mathbb{A}[x_r]$ by a $q \in \mathbb{Q}[x_r]$ in the extension phase of the CAD procedure (cf. Section 3.3.4).

The implementation of cylindrical algebraic decomposition is by far not as optimized as the REDLOG system. Therefore, we generally expect that the use of REDUCE instead of our own cylindrical algebraic decomposition has a (probably big) performance advantage. The disadvantage of using REDUCE is mainly that there is some overhead in passing formulas to REDUCE and reading back the results into our implementation's data structures. See Chapter 5 for selected results of our experiments.

[1] The implementation does not require the computation of minimal polynomials (since this would require an algorithm for polynomial factorization which we did not implement), because the arithmetic is performed in the ring $\mathbb{Q}[x]/(\delta)$ where δ is a defining polynomial for an algebraic number α (and hence a multiple of μ); additionally, the knowledge of an isolation interval for α is used to simulate the arithmetic of the field $\mathbb{Q}[x]/(\mu)$ in the ring $\mathbb{Q}[x]/(\delta)$.

Chapter 4

Applying Quantifier Elimination to the Polyhedron Model

The algorithms used in the context of the polyhedron model describe transformations on polyhedra. That means that they usually take one polyhedron or several polyhedra as input an return a new polyhedron or some new polyhedra. The most elementary operation which many algorithms have to perform on a polyhedron is to solve one of the inequalities describing the polyhedron for a variable x. As has been outlined in Section 2.1.4, there is (from the algorithmic point of view) a difference between fixed real coefficients and coefficients depending on parameters: if the value of the coefficient of x is known, its sign is also known; if, on the other hand, the coefficient depends on parameters, its sign (possibly) depends on the parameters, and the result of the computation has to include a case distinction on the sign of the parameter. Therefore, if we generalize the algorithms of the polyhedron model to allow non-linear parameters, they have to be changed in the sense that they will have to return not a single polyhedron as before, but case distinctions consisting of conditions (in the parameters) and results valid for a given condition.

These case distinctions can be represented in different ways. Two very basic possibilities are

- a list (or set) of condition/result pairs,
- a decision tree, carrying conditions at its inner nodes (or its edges) and results at its leaf nodes.

Quantifier elimination with answer (as described in Section 3.2.5) delivers a set of condition/result pairs. In most cases, however, we prefer the tree representation of results generated by the generalized algorithms. We make this choice because tree representations have some advantages for our purposes:

- Decision trees are easily constructed when performing a generalized version of algorithms depending on the sign of coefficients, like Fourier-Motzkin elimination.
- Using a *lazy* language for the implementation of the generalized algorithm, the algorithm itself and a top-down tree simplifier can be implemented as separate modules.

Our implementation language is Haskell [PJ03], a lazy, purely functional programming language. We present some of the algorithms and data structures in this section in a Haskell-like notation. Therefore, we assume that the reader is familiar with Haskell. An introduction to Haskell can be found in [HPF00].

In this chapter we first define the possible input formats for the generalized polyhedron model (Section 4.1). We then introduce a representation of the decision trees (Section 4.1.1) we use to represent the results of generalized algorithms. We develop a general "recipe" to generalize existing algorithms to handle non-linear parameters (Section 4.2). As an example for a generalized algorithm constructed using the recipe, we present a Fourier-Motzkin algorithm for non-linear parameters (Section 4.2.1). After that, we are taking a different approach: we develop new algorithms using quantifier elimination (with and without answer), mainly an algorithm equivalent to a generalized Fourier-Motzkin (Section 4.3.2) and algorithms for computing unions of polyhedra (Section 4.3.3).

4.1 The Generalized Polyhedron Model

As is outlined in Section 2.1.4, we deal with inequalities of the form

$$\sum_{i=1}^{n} c_i \cdot x_i + d \geq 0 \tag{4.1}$$

where $x_1, \ldots, x_n$ are variables and $c_1, \ldots, c_n, d$ are the coefficients of the inequality which may depend on the parameters $p_1, \ldots, p_m$. In general, we have to choose $\mathbb{Q}(p_1, \ldots, p_m)$ as domain for $c_1, \ldots, c_n, d$ since we want to divide by (non-zero) coefficients to solve for a variable. $\mathbb{Q}(p_1, \ldots, p_m)$ is a field and therefore closed under division (by non-zero divisors).

The disadvantage of using $\mathbb{Q}(p_1, \ldots, p_m)$ as domain for the coefficients is that Inequality (4.1) is not a Σ_{ord}-formula. Some algorithms which we present here require the input to be given as Σ_{ord}-formulas. We can always write Inequality (4.1) equivalently as a Σ_{ord}-formula (under the assumption that the denominators of the coefficients are non-zero). We calculate the least common multiple l of the denominators of $c_1, \ldots, c_n, d$. l is a polynomial from $\mathbb{Q}[p_1, \ldots, p_m]$ and if we multiply Inequality (4.1) by l or l^2, all the denominators cancel out and the resulting inequality is a Σ_{ord}-formula. If we can determine that $l > 0$ holds under the assumptions we make about the parameters, it suffices to multiply Inequality (4.1) with l, otherwise we must use l^2 since l^2 is always greater than zero. If we rewrite an equation (or an inequality with the relation $\neq$), it suffices to use l as multiplier in any case since the sign of l is irrelevant in this case.

Whether the assumptions about the parameters ensure that $l > 0$ holds can be decided using quantifier elimination. Assume that the quantifier-free formula χ expresses the assumptions we make. We then decide the formula

$$\forall p_1 \cdots \forall p_m \, (\chi \to l > 0)$$

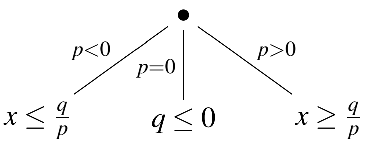
in $\underline{\mathbb{R}}$, and the truth value of this formula tells us whether the assumptions imply that l is greater than zero or not. In the following, we will not talk about the conversion of coefficients form $\mathbb{Q}(p_1, \ldots, p_m)$ to $\mathbb{Q}[p_1, \ldots, p_m]$ explicitly, we assume that the inequality systems are transformed before applying one of our algorithms if necessary.

4.1.1 Tree Representation of Case Distinctions

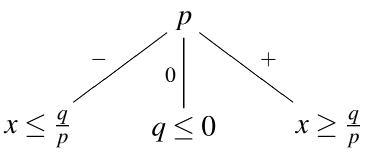
To show the code for a simple decision tree, let us take $p \cdot x - q \geq 0$ as an example. If we solve $p \cdot x - q \geq 0$ for x we get this case distinction:

if $p < 0$ **then**
  $x \leq \frac{q}{p}$
else if $p = 0$ **then**
  $q \leq 0$
else if $p > 0$ **then**
  $x \geq \frac{q}{p}$
end if

This case distinction could be represented by a tree carrying the results (like $x \leq \frac{q}{p}$) at the leaves and the conditions (like $p < 0$) at the edges,

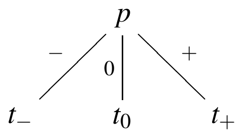


but, since all the conditions ($p < 0$, $p = 0$, $p > 0$) share the same polynomial p, we represent the case distinction by a node carrying the polynomial p and having sub-trees for each of the cases < 0, $= 0$, > 0. As abbreviations we write $-$ for < 0, 0 for $= 0$, and $+$ for > 0:



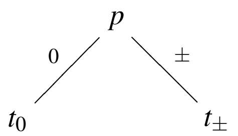
Special cases of this ternary case distinction occur, when two of the three cases share the same sub-tree. Therefore, our tree data type has three different constructors for inner nodes of the tree:

- *SCond* p t_- t_0 t_+

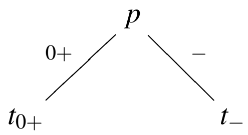
  t_- for $p < 0$, t_0 for $p = 0$, and t_+ for $p > 0$.

- *EqCond* p t_0 $t_\pm$

  t_0 for $p = 0$ and $t_\pm$ for $p \neq 0$

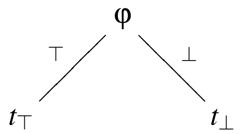
- *GeCond* p t_{0+} t_-

  t_{0+} for $p \geq 0$ and t_- for $p < 0$

Another case which can occur (especially when we want to represent the result of an external tool as a tree) is that we do not want to make a distinction on the sign of a polynomial but use an arbitrary quantifier-free formula to discriminate two cases. For this purpose, we add a fourth constructor:

- *FCond* φ $t_\top$ $t_\bot$



$t_\top$ when φ is true and $t_\bot$ when φ is false.

In summary, the definition of our decision tree datatype as an algebraic data type in a Haskell-like notation is:

$$
\begin{aligned}
\textbf{data}\ \mathit{Tree}\ \alpha = {} & \mathit{Leaf}\ \alpha \\
\mid {} & \mathit{SCond}\quad \mathit{Polynomial}\ (\mathit{Tree}\ \alpha)\ (\mathit{Tree}\ \alpha)\ (\mathit{Tree}\ \alpha) \\
\mid {} & \mathit{EqCond}\ \mathit{Polynomial}\ (\mathit{Tree}\ \alpha)\ (\mathit{Tree}\ \alpha) \\
\mid {} & \mathit{GeCond}\ \mathit{Polynomial}\ (\mathit{Tree}\ \alpha)\ (\mathit{Tree}\ \alpha) \\
\mid {} & \mathit{FCond}\quad \mathit{QfFormula}\ (\mathit{Tree}\ \alpha)\ (\mathit{Tree}\ \alpha)
\end{aligned}
$$

The decision trees which can be constructed with these five constructors always represent a *complete* case distinction, since every constructor for inner nodes holds sub-trees for every possible case. When incomplete case distinctions are required, we use trees carrying values of type *Maybe* α instead of α

$$
\begin{aligned}
\textbf{data}\ \mathit{Maybe}\ \alpha = {} & \mathit{Nothing} \\
\mid {} & \mathit{Just}\ \alpha
\end{aligned}
$$

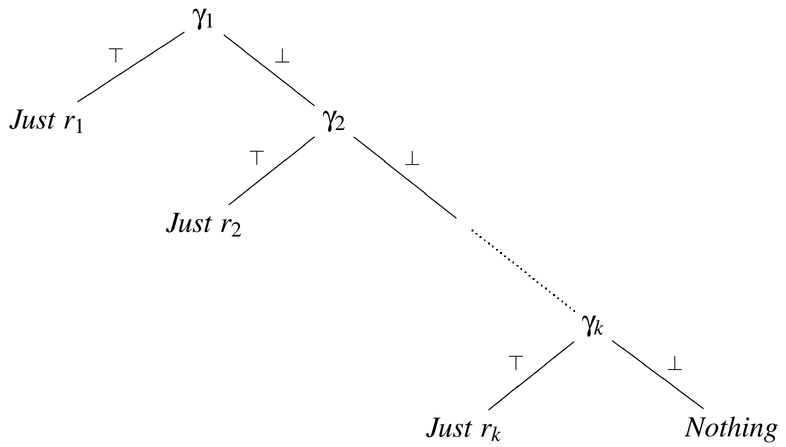
where the value *Nothing* is used to express the notion of "no solution."

4.1.2 Representing Results from Quantifier Elimination with Answer as Trees

The node constructor *FCond* is mainly used to represent the result of a quantifier elimination with answer as a decision tree. As noted in Section 3.2.5, the result of a quantifier elimination with answer is a set

$$\{(\gamma_1, r_1), \ldots, (\gamma_k, r_k)\}$$

where each γ_i represents a condition under which the answer r_i (which is a list of substitutions, i.e., values for the existentially quantified variables) is a solution of the input formula. We transform this result into the following tree:



Note that the tree is carrying values of type *Maybe Substitution* instead of just *Substitution*, since a decision tree always represents a complete case distinction and we use the value *Nothing* to stand for "no solution" in the case $\neg\gamma_1 \wedge \cdots \wedge \neg\gamma_k$.

A formal description of this transformation process is given by the following Haskell code using the *foldr* combinator, which performs a right-to-left list reduction:

$$
\begin{aligned}
&\mathit{qeaToTree} \qquad\qquad :: [(\mathit{Formula}, \alpha)] \to \mathit{Tree}\ (\mathit{Maybe}\ \alpha) \\
&\mathit{qeaToTree}\ \ \mathit{qeaRes} = \mathit{foldr}\ \ \mathit{combine}\ \ (\mathit{Leaf\ Nothing})\ \ \mathit{qeaRes} \\
&\quad \textbf{where} \\
&\qquad \mathit{combine}\ \ (\gamma, r)\ \ \mathit{subtree}\ = \ \mathit{FCond}\ \ \gamma\ \ (\mathit{Leaf}\ r)\ \ \mathit{subtree}
\end{aligned}
$$

The function *qeaToTree* works not only with substitutions as results, but for arbitrary types since we will occasionally transform the result (e.g., to inequalities) before changing the structure to the tree representation.

4.2 Recipe for the Generalization of Algorithms to Non-linear Parameters

As is outlined in Section 2.1.4, the fundamental operation of many algorithms used in the polyhedron model is solving an equation or an inequality for a given variable. Since the signum of the coefficient of the variable determines how the respective equation or inequality is handled, algorithms usually contain constructs like

$$
\begin{aligned}
&\textbf{case}\ \mathit{signum}\ c\ \textbf{of} \\
&\quad \mathit{Negative} \to t_- \\
&\quad \mathit{Zero} \qquad\ \to t_0 \\
&\quad \mathit{Positive}\ \ \to t_+
\end{aligned}
$$

The signum of c is determined statically by the input of the algorithm since the coefficient is a fixed real (usually even rational) number. Depending on this signum, the computation continues with either of the subprograms t_-, t_0, or t_+.

In a generalized version of the algorithm, the coefficients are rational functions in the parameters, i.e., $c \in \mathbb{Q}(p_1, \ldots, p_n)$. Then c can be represented by a fraction of polynomials, $c = \frac{c_1}{c_2}$ for $c_1, c_2 \in \mathbb{Q}[p_1, \ldots, p_n]$, $c_2 \neq 0$. The signum of c is then equal to the signum of the polynomial $c_1 \cdot c_2$. Therefore, the case distinction on the sign of c is transformed into the tree node

$$\mathit{SCond}\ (c_1 \cdot c_2)\ t_-\ t_0\ t_+$$

If the case distinction is only binary, i.e., $c = 0$ vs. $c \neq 0$ or $c \geq 0$ vs. $c < 0$, then one of the other constructors (*EqCond* or *GeCond*) can be used to avoid the duplication of subtrees. This rewriting rule replaces all case distinctions in the *code* of the algorithm by case distinctions in the result *data structure*.

Obviously, some other changes in the implementation of the algorithm are necessary. When a final result z is returned, the correct tree representation for z is *Leaf* z. The type α of the result has to be changed to *Tree* α. Finally, if a result z (of former type α) is used as the argument of another function, i.e., $f\ z$, then the application of f has to be “lifted” correctly to the whole tree using a higher-order combinator. We have to distinguish two cases here:

1. If the function f works without modification on the generalized input, i.e., the result type β of f need not be changed to *Tree* β, then a combinator usually called *fmap* in Haskell is appropriate to apply f to all the leaves of a tree with a call *fmap* $f\ z$ instead of the original $f\ z$:

$$\begin{array}{ll}
\mathit{fmap} & :: (\alpha \to \beta) \to \mathit{Tree}\ \alpha \to \mathit{Tree}\ \beta \\
\mathit{fmap}\ f\ (\mathit{Leaf}\ z) & = \mathit{Leaf}\ (f\ z) \\
\mathit{fmap}\ f\ (\mathit{SCond}\ p\ t_-\ t_0\ t_+) & = \mathit{SCond}\ p\ (\mathit{fmap}\ f\ t_-)\ (\mathit{fmap}\ f\ t_0)\ (\mathit{fmap}\ f\ t_+) \\
\mathit{fmap}\ f\ (\mathit{EqCond}\ p\ t_0\ t_\pm) & = \mathit{EqCond}\ p\ (\mathit{fmap}\ f\ t_0)\ (\mathit{fmap}\ f\ t_\pm) \\
\mathit{fmap}\ f\ (\mathit{GeCond}\ p\ t_{0+}\ t_-) & = \mathit{GeCond}\ p\ (\mathit{fmap}\ f\ t_{0+})\ (\mathit{fmap}\ f\ t_-) \\
\mathit{fmap}\ f\ (\mathit{FCond}\ \varphi\ t_\top\ t_\bot) & = \mathit{FCond}\ \varphi\ (\mathit{fmap}\ f\ t_\top)\ (\mathit{fmap}\ f\ t_\bot)
\end{array}$$

2. If the function f has to be generalized such that its result type changes from β to *Tree* β, the application of f using the *fmap* combinator, like in *fmap* $f\ z$, would yield a result of type *Tree* (*Tree* β). Using a different combinator we can “flatten” this tree of trees into a single tree:

$$\begin{array}{ll}
\mathit{treeMap} & :: (\alpha \to \mathit{Tree}\ \beta) \to \mathit{Tree}\ \alpha \to \mathit{Tree}\ \beta \\
\mathit{treeMap}\ f\ (\mathit{Leaf}\ z) & = f\ z \\
\mathit{treeMap}\ f\ (\mathit{SCond}\ p\ t_-\ t_0\ t_+) & = \mathit{SCond}\ p\ (\mathit{treeMap}\ f\ t_-)\ (\mathit{treeMap}\ f\ t_0)\ (\mathit{treeMap}\ f\ t_+) \\
\mathit{treeMap}\ f\ (\mathit{EqCond}\ p\ t_0\ t_\pm) & = \mathit{EqCond}\ p\ (\mathit{treeMap}\ f\ t_0)\ (\mathit{treeMap}\ f\ t_\pm) \\
\mathit{treeMap}\ f\ (\mathit{GeCond}\ p\ t_{0+}\ t_-) & = \mathit{GeCond}\ p\ (\mathit{treeMap}\ f\ t_{0+})\ (\mathit{treeMap}\ f\ t_-) \\
\mathit{treeMap}\ f\ (\mathit{FCond}\ \varphi\ t_\top\ t_\bot) & = \mathit{FCond}\ \varphi\ (\mathit{treeMap}\ f\ t_\top)\ (\mathit{treeMap}\ f\ t_\bot)
\end{array}$$

   The difference between *fmap* and *treeMap* is in the base case, where *treeMap* does not produce a leaf carrying the result $f\ z$, but simply returns the tree produced by $f\ z$.

4.2.1 A Generalized Fourier-Motzkin Algorithm

The Fourier-Motzkin algorithm, as described in Section 2.1.2, can only handle inequality systems with constant coefficients of the variables. This restriction is inherently built into the algorithm, since the construction of the sets L_i and U_i of lower and upper bounds depends on the signs of the coefficients of x_i.

The generalized version of the Fourier-Motzkin algorithm presented here works as follows. Whenever the inequalities are to be solved for a variable x_i, then a case distinction on the signs of all terms appearing as coefficients of x_i is made and the sets L_i and U_i are determined for each of the cases. In other words, this is an application of the recipe (Section 4.2) for generalizing algorithms to non-linear parameters. We do not start with an implementation of the Fourier-Motzkin algorithm and apply formally the recipe from Section 4.2 to it, since this produces clumsy code. We just follow the idea and give a more readable implementation of the algorithm in pseudo-Haskell, which takes a set of inequalities S and the number of variables n as input (Figure 4.1).

$$
\begin{array}{l}
\mathit{fourierMotzkin} \quad :: \mathit{Int} \rightarrow \mathit{Set\ Inequality} \rightarrow \\
\qquad\qquad\qquad\quad \mathit{Tree}\ \big(\mathit{Maybe}\ [(\mathit{Set\ Inequality},\ \mathit{Set\ Inequality})]\big) \\
\mathit{fourierMotzkin}\ n\ S = \mathit{fourier}\ n\ S\ [\,] \\
\\
\mathit{fourier} \quad :: \mathit{Int} \rightarrow \mathit{Set\ Inequality} \rightarrow [(\mathit{Set\ Inequality},\ \mathit{Set\ Inequality})] \rightarrow \\
\qquad\qquad\quad \mathit{Tree}\ \big(\mathit{Maybe}\ [(\mathit{Set\ Inequality},\ \mathit{Set\ Inequality})]\big) \\
\mathit{fourier}\ 0\ S_0\ \mathit{LUs} = \mathit{FCond}\ \ (\bigwedge S_0)\ \ \big(\mathit{Leaf}\ (\mathit{Just\ LUs})\big)\ \ (\mathit{Leaf\ Nothing}) \\
\mathit{fourier}\ n\ S_n\ \mathit{LUs} = \mathit{eliminate}\ \ S_n\ \ \varnothing\ \ \varnothing\ \ \varnothing \\
\quad \textbf{where} \\
\quad\quad \mathit{eliminate}\ \ \varnothing\ \ S_{new}\ \ L\ \ U\ \ = \\
\quad\quad\quad \mathit{fourier}\ \ (n-1)\ \ (S_{new} \cup S'_{new})\ \ (\mathit{LUs} +\!\!+ [(L,U)]) \\
\quad\quad\quad \textbf{where} \\
\quad\quad\quad\quad S'_{new}\ = \{t' - t \geq\ 0 \mid (x \geq\ t) \in\ L,\ (x \leq\ t') \in\ U\} \\
\quad\quad \mathit{eliminate}\ \ (S \dot\cup \{\frac{t_1}{t_2} \cdot\ x_n + t' \geq\ 0\})\ \ S_{new}\ \ L\ \ U\ \ = \\
\quad\quad\quad \mathit{SCond}\ (t_1 \cdot\ t_2) \\
\quad\quad\quad\quad (\mathit{eliminate}\ \ S\ \ S_{new}\ \ L\ \ (U \cup \{x_n \leq\ -\frac{t' \cdot t_2}{t_1}\})) \\
\quad\quad\quad\quad (\mathit{eliminate}\ \ S\ \ (S_{new} \cup \{t' \geq\ 0\})\ \ L\ \ U\ ) \\
\quad\quad\quad\quad (\mathit{eliminate}\ \ S\ \ S_{new}\ \ (L \cup \{x_n \geq\ -\frac{t' \cdot t_2}{t_1}\})\ \ U\ )
\end{array}
$$

Figure 4.1: Generalized Fourier-Motzkin elimination for non-linear parameters.

The algorithm works like the original Fourier-Motzkin algorithm, except that it has to make case distinctions when constructing the sets of lower and upper bounds. The third parameter of function *fourier* serves as an accumulator for the upper and lower bounds which have already been found. The construction of the case distinction is done by the function *eliminate*. It handles recursively every inequality from the set S_n passed to it as its first parameter, and makes a case distinction on whether the selected inequality represents a lower bound, an upper bound, or no bound for the highest remaining variable x_n. The second, third, and fourth argument of *eliminate* are accumulators for the already found lower bounds, non-bounds, and upper bounds, respectively. When all inequalities have been analyzed (and the first parameters becomes the empty set), *eliminate* constructs the projection of S_n to the lower dimensions ($S_{new} \cup S'_{new}$) and passes this projection, together with the updated accumulator of lower and upper bounds, to a recursive invocation of *fourier*.

When *fourier* is called with argument 0, the procedure has eliminated every variable and the set S_0 contains conditions in the parameters only. The accumulator *LUs* has collected all the lower and upper bounds, i.e., $LUs = [(L_1,U_1),\ldots,(L_n,U_n)]$, where L_i and U_i are the sets of lower and upper bounds for x_i. The input inequality system is feasible (in the particular case determined by the cast distinction constructed by *eliminate* "above" this base case) if and only if $\bigwedge S_0$ holds. Therefore, we guard the solution *Just LUs* with the condition $\bigwedge S_0$ (and provide the value *Nothing*, for "no solution", in the other case).

In case we are not interested in when the given inequality system S is feasible (in the reals) and when not, but are interested in a (possibly empty) description of the points in the polyhedron described by S, we need not make the case distinction on $\bigwedge S_0$ in the base case of *fourier*. Then it is sufficient to use

$$\mathit{fourier}\ 0\ S_0\ \mathit{LUs} = \mathit{Leaf}\ (\mathit{Just}\ \mathit{LUs})$$

as base case for the recursion in *fourier* and ignore the conditions in S_0.

Not shown in Figure 4.1, but part of our implementation, is that Fourier-Motzkin can be significantly sped up by exploiting equations if they are present in the input. The implementation of this is fairly simple. Assume we have an equation $c \cdot x_n = e$ for some linear expression e with variables $x_1, \ldots, x_{n-1}$. Before we eliminate x_n using the actual Fourier-Motzkin method, we make a case distinction on whether c is zero or not. If c is not zero, we can substitute $\frac{e}{c}$ for x_n in the inequalities and the equations. $\frac{e}{c}$ is then a lower and upper bound for x_n. If c is zero, we replace the equation $c \cdot x_n = e$ by $0 = e$ and check the other equations (or perform Fourier-Motzkin elimination for x_n if no further equations are available).

Note that the results computed by Fourier-Motzkin elimination often contain superfluous bounds. For example, it can easily happen that the lower bounds for x_2 contain both $x_2 \geq 2x_1 + p$ and $x_2 \geq 2x_1 + p + 3$. In this case, the bound $2x_1 + p$ is superfluous, since it is dominated (for every value of the parameter p) by the bound $2x_1 + p + 3$. Another example is $x_2 \geq 2x_1$ and $x_2 \geq 2x_1 - p$ if we know that either $p \geq 0$ or $p \leq 0$; e.g., in the case $p \geq 0$, the only relevant bound is $x_2 \geq 2x_1$. Our implementation contains some optimizations for the bound sets which exploits such situations. We describe these optimizations here for lower bounds; it is analogous for upper bounds.

We check, for two given lower bounds l_1 and l_2, if $d := l_1 - l_2$ is independent of the variables, i.e., if $d \in \mathbb{Q}(p_1, \ldots, p_n)$. If this is not the case, we leave both l_1 and l_2 in the set of bounds and go on checking other combinations of bounds. If $d \in \mathbb{Q}(p_1, \ldots, p_n)$, we check whether $d \in \mathbb{Q}$, that is, whether the difference of l_1 and l_2 is a constant. If $d \in \mathbb{Q}$ and $d \geq 0$, then l_1 dominates l_2 and we can remove l_2 from the set of lower bounds. If $d \leq 0$, we can remove l_1. When $d \notin \mathbb{Q}$, then we check if the context C for the part of the decision tree we are in implies that $d \geq 0$ or $d \leq 0$ holds. This is, of course, solved by deciding the problems

$$\underline{\mathbb{R}} \models \forall\big(\bigwedge C \to d \geq 0\big)$$

and

$$\underline{\mathbb{R}} \models \forall\big(\bigwedge C \to d \leq 0\big)$$

using a quantifier elimination method. If the context implies one of the two conditions, either l_2 or l_1 can be removed from the set of lower bounds.

Obviously, comparing every lower (upper) bound against every other lower (upper) bound gives rise to a quadratic amount of comparisons in the number of lower (upper) bounds. The experiments we conducted with our implementation showed that, for simple examples, this causes a moderate slowdown but, for more complex examples, the optimization is vital for obtaining a result at all.

4.2.2 Simplifying the Decision Trees

The previous section shows that a naive application of the recipe for generalizing algorithms to non-linear parameters leads to huge decision trees with a lot of superfluous case distinctions. Since we use a lazy implementation language, it is not necessary to build simplification algorithms into the generalized algorithms. Instead, the simplification of the decision trees can be implemented completely separately from the generalized algorithm, since the laziness of the implementation language guarantees that irrelevant parts of the decision tree are never computed.

We implement a top-down simplification method. This method starts with some assumptions on the parameter values, represented as logical formulas. For example, $p \geq 0$ is a common assumption. These assumptions are called the *context* with respect to which the simplification is performed. The actual simplification begins at the root of the decision tree. At every node which represents a case distinction, it is checked whether the context implies one of the conditions which make a certain sub-tree of the node applicable. If the context implies one of these conditions, the node is replaced by the respective sub-tree and the simplification continues on this sub-tree. If the context does not imply any of the conditions, then the node is retained and the simplification is performed on each sub-tree after the condition required for the sub-tree to be applicable is added to the context for the simplification of the sub-tree.

As an example consider the simplification of the node $n = (SCond\; p\; t_-\; t_0\; t_+)$ under a context C. If the logical formula $\bigwedge C \rightarrow p > 0$ holds in $\underline{\mathbb{R}}$, it is clear that, under the assumption that C holds, the sub-trees t_- and t_0 of n are irrelevant and the node n can be replaced by t_+. The same simplification can be performed if $\bigwedge C \rightarrow p = 0$ or $\bigwedge C \rightarrow p < 0$ holds. If, for example, $\bigwedge C \rightarrow p > 0$ and $\bigwedge C \rightarrow p = 0$ do not hold, but $\bigwedge C \rightarrow p \geq 0$ holds, the ternary constructor *SCond* can be replaced by the simpler *EqCond*: since t_- is irrelevant, n can be expressed by the binary case distinction $EqCond\; p\; t_0\; t_+$. Similar optimizations can be performed in some other cases; the complete simplification procedure is shown in a pseudo-Haskell notation in Figure 4.2.

The simplification procedure in Figure 4.2 uses a predicate *implies* and a function *simplifyFormula* not shown in the figure. The predicate *implies* is a function whose specification is given by

$$
\begin{array}{ll}
\mathit{implies} & :: \mathit{Set\ Formula} \rightarrow \mathit{Formula} \rightarrow \mathit{Bool} \\
\mathit{implies\ context}\ \varphi = \textbf{if}\ \big(\underline{\mathbb{R}} \models \forall(\bigwedge \mathit{context} \rightarrow \varphi)\big)\ \textbf{then}\ \mathit{True}\ \textbf{else}\ \mathit{False} &
\end{array}
$$

implies takes care of deciding whether a set of formulas (the context *context*) logically implies another formula φ in $\underline{\mathbb{R}}$. The notation $\forall(\bigwedge context \rightarrow \varphi)$ is used to express that the formula $\bigwedge context \rightarrow \varphi$ is to be prefixed with a universal quantifier for every free variable of $\bigwedge context \rightarrow \varphi$.

To decide $\forall(\bigwedge context \rightarrow \varphi)$, a quantifier elimination method like virtual substitution (Section 3.2) or (since the formula has no free variables) a decision method like cylindrical algebraic decomposition (Section 3.3) can be used. It is obvious, that when simplifying a *SCond* node, very similar formulas have to be decided, namely $\bigwedge context \rightarrow p\,\rho\, 0$ for different relations $\rho \in \{<, \leq, =, \neq, \geq, >\}$. The simplification procedure has been chosen such that at most 3 different formulas have to be decided (of the possible 6 ones). When we use REDLOG to implement the *implies* function, we have to make these similar calls to the quantifier elimination procedure. But with our CAD implementation we need not repeat the whole work thrice: The terms in the formulas are the same for each of the six cases, so the

$$
\begin{array}{l}
\mathit{simplify} \quad :: \mathit{Set\ Formula} \to \mathit{Tree}\ \alpha \to \mathit{Tree}\ \alpha \\
\mathit{simplify\ context}\ (\mathit{Leaf}\ z) = \mathit{Leaf}\ z \\
\mathit{simplify\ context}\ (\mathit{SCond}\ p\ t_-\ t_0\ t_+) = \\
\quad \textbf{if}\ \mathit{implies\ context}\ (p \geq 0)\ \textbf{then} \\
\qquad \textbf{if}\ \mathit{implies\ context}\ (p > 0)\ \textbf{then}\ \mathit{simplify\ context}\ t_+ \\
\qquad \textbf{else if}\ \mathit{implies\ context}\ (p = 0)\ \textbf{then}\ \mathit{simplify\ context}\ t_0 \\
\qquad \textbf{else}\ \mathit{EqCond}\ p \big(\mathit{simplify}\ (\mathit{context} \cup \{p = 0\})\ t_0\big) \\
\qquad\qquad \big(\mathit{simplify}\ (\mathit{context} \cup \{p > 0\})\ t_+\big) \\
\quad \textbf{else if}\ \mathit{implies\ context}\ (p \leq 0)\ \textbf{then} \\
\qquad \textbf{if}\ \mathit{implies\ context}\ (p < 0)\ \textbf{then}\ \mathit{simplify\ context}\ t_- \\
\qquad \textbf{else}\ \mathit{EqCond}\ p \big(\mathit{simplify}\ (\mathit{context} \cup \{p = 0\})\ t_0\big) \\
\qquad\qquad \big(\mathit{simplify}\ (\mathit{context} \cup \{p < 0\})\ t_-\big) \\
\quad \textbf{else if}\ \mathit{implies\ context}\ (p \neq 0)\ \textbf{then}\ \mathit{GeCond}\ p \big(\mathit{simplify}\ (\mathit{context} \cup \{p > 0\})\ t_+\big) \\
\qquad\qquad \big(\mathit{simplify}\ (\mathit{context} \cup \{p < 0\})\ t_-\big) \\
\quad \textbf{else}\ \mathit{SCond}\ p \big(\mathit{simplify}\ (\mathit{context} \cup \{p < 0\})\ t_-\big) \\
\qquad\qquad \big(\mathit{simplify}\ (\mathit{context} \cup \{p = 0\})\ t_0\big) \\
\qquad\qquad \big(\mathit{simplify}\ (\mathit{context} \cup \{p > 0\})\ t_+\big) \\
\mathit{simplify\ context}\ (\mathit{EqCond}\ p\ t_0\ t_\pm) = \\
\quad \textbf{if}\ \mathit{implies\ context}\ (p = 0)\ \textbf{then}\ \mathit{simplify\ context}\ t_0 \\
\quad \textbf{else if}\ \mathit{implies\ context}\ (p \neq 0)\ \textbf{then}\ \mathit{simplify\ context}\ t_\pm \\
\quad \textbf{else}\ \mathit{EqCond}\ p \big(\mathit{simplify}\ (\mathit{context} \cup \{p = 0\})\ t_0\big) \\
\qquad\qquad \big(\mathit{simplify}\ (\mathit{context} \cup \{p \neq 0\})\ t_\pm\big) \\
\mathit{simplify\ context}\ (\mathit{GeCond}\ p\ t_{0+}\ t_-) = \\
\quad \textbf{if}\ \mathit{implies\ context}\ (p \geq 0)\ \textbf{then}\ \mathit{simplify\ context}\ t_{0+} \\
\quad \textbf{else if}\ \mathit{implies\ context}\ (p < 0)\ \textbf{then}\ \mathit{simplify\ context}\ t_- \\
\quad \textbf{else}\ \mathit{GeCond}\ p \big(\mathit{simplify}\ (\mathit{context} \cup \{p \geq 0\})\ t_{0+}\big) \\
\qquad\qquad \big(\mathit{simplify}\ (\mathit{context} \cup \{p < 0\})\ t_-\big) \\
\mathit{simplify\ context}\ (\mathit{FCond}\ \varphi\ t_\top\ t_\bot) = \\
\quad \textbf{if}\ \mathit{implies\ context}\ \varphi\ \textbf{then}\ \mathit{simplify\ context}\ t_\top \\
\quad \textbf{else if}\ \mathit{implies\ context}\ (\neg\varphi)\ \textbf{then}\ \mathit{simplify\ context}\ t_\bot \\
\quad \textbf{else}\ \mathit{FCond}\ \psi \big(\mathit{simplify}\ (\mathit{context} \cup \{\psi\})\ t_{0+}\big) \\
\qquad\qquad \big(\mathit{simplify}\ (\mathit{context} \cup \{\neg\psi\})\ t_-\big) \\
\quad \textbf{where} \\
\qquad \psi = \mathit{simplifyFormula\ context}\ \varphi
\end{array}
$$

Figure 4.2: Top-down simplifying algorithm for decision trees

test points computed by the CAD procedure are the same. We only calculate the test points once, keep only the points where $\bigwedge \mathit{context}$ is true (since if $\bigwedge \mathit{context}$ is false, the implication $\bigwedge \mathit{context} \rightarrow p \,\rho\, 0$ a is true anyway), and check the sign of p for the remaining points.

The function *simplifyFormula* is used to simplify a quantifier-free formula in the context of a set of formulas. For example, *simplifyFormula* $\{p > 0, q \leq 4\} \big(p \geq 1 \wedge (q \leq 3 \vee q \geq 5)\big)$ could yield the simpler formula $q \leq 3$. We do not give a simplification algorithm here; good algorithms can be found in the literature, e.g., in [DS97b]. The formula simplification algorithm may require the context to be given as a set of *atomic* formulas. In this case, *simplifyFormula* should take care of replacing conjunctions of atomic formulas by the individual formulas and dropping additional formulas from the context before passing the context to the simplification algorithm.

4.3 New Algorithms based on Quantifier Elimination

The previous section focuses on generalizing an existing algorithm of the polyhedron model to non-linear parameters. In this approach, quantifier elimination (or a decision method) is used to reduce the size of decision trees by detecting irrelevant branches. In this section, we discuss some direct applications of quantifier elimination, where we use quantifier elimination with answer to calculate some desired results instead of generalizing an existing algorithm of the polyhedron model. Of course, this requires that the problem we wish to solve can be expressed as a first-order formula in $\mathbb{R}$.

We show how the lexicographic minimum of a polyhedron can be found (Section 4.3.1), how a sorting of an inequality system, which is equivalent to Fourier-Motzkin elimination, can be computed using quantifier elimination (Section 4.3.2), and how to find convex and disjoint unions of polyhedra (Section 4.3.3).

4.3.1 Lexicographic Minima and Maxima

The calculation of lexicographic minima and maxima plays an important role in optimization problems in the polyhedron model.

The procedure for calculating lexicographic minima and maxima differs only in the orientation of some relation symbols in the input formula. Therefore, we only show how to deal with the lexicographic minimum.

The lexicographic minimum of a polyhedron P is a point $x \in P$ inside the polyhedron which is lexicographically less than or equal to every point $y \in P$. We denote "lexicographically less than or equal" with the symbol $\preceq$. The signature Σ_{ord} does not contain a symbol for lexicographic ordering, but the usual recursive definition of $(a_1, \ldots, a_n) \preceq (b_1, \ldots, b_n)$ gives, in fact, a Σ_{ord}-formula for lexicographic ordering, and we use the notation $(a_1, \ldots, a_n) \preceq (b_1, \ldots, b_n)$ as a short hand for the first-order formula defined as follows:

$$
\begin{aligned}
a_1 \preceq b_1 &:= a_1 \leq b_1 \\
(a_1, \ldots, a_n) \preceq (b_1, \ldots, b_n) &:= a_1 < b_1 \vee \big(a_1 = b_1 \wedge (a_2, \ldots, a_n) \preceq (b_2, \ldots, b_n)\big) \quad \text{(for } n \geq 2\text{)}
\end{aligned}
$$

We are now able to express that $(x_1, \ldots, x_n)$ is the lexicographic minimum of a polyhedron P as a first-order formula. Assume that P is defined by the first-order formula φ (usually φ is a conjunction of linear inequalities) and $\mathit{frvar}(\varphi) = \{x_1, \ldots, x_n, p_1, \ldots, p_m\}$, where $x_1, \ldots, x_n$

are the variables describing the dimensions of the polyhedron and $p_1, \ldots, p_m$ are the parameters the polyhedron is parametrized with. Let, for some new variables $y_1, \ldots, y_n$,

$$\psi \; := \; \varphi \wedge \forall y_1 \cdots \forall y_n \left(\varphi[x_1/y_1, \ldots, x_n/y_n] \rightarrow (x_1, \ldots, x_n) \preceq (y_1, \ldots, y_n) \right)$$

This formula expresses that $(x_1, \ldots, x_n)$ is a point in P (because of the φ conjunct) and that, if $(y_1, \ldots, y_n)$ is any point in P, then $(x_1, \ldots, x_n)$ is lexicographically less than or equal to $(y_1, \ldots, y_n)$.

To get values for $x_1, \ldots, x_n$ (which are uniquely determined if they exist), we apply the generalized quantifier elimination to the following formula:

$$\exists x_1 \cdots \exists x_n \, \psi$$

The result is (as described in Section 3.2.5) a set

$$\begin{aligned} \{ & (\gamma_1, \{x_1 = t_{1,1}, \ldots, x_n = t_{1,n}\}), \\ & \cdots, \\ & (\gamma_k, \{x_1 = t_{k,1}, \ldots, x_n = t_{k,n}\}) \} \end{aligned}$$

of conditions γ_i and the respective lexicographic minimum $(t_{i,1}, \ldots, t_{i,n})$. The formula γ_i and the terms $t_{i,1}, \ldots, t_{i,n}$ contain the parameters $p_1, \ldots, p_m$ as variables. We can transform this result set into a tree for further use as presented in Section 4.1.2.

4.3.2 Sorting a System of Inequalities

The Fourier-Motzkin algorithm, as defined in Section 2.1.2, solves the problem of sorting an inequality system. Unfortunately, the Fourier-Motzkin elimination is (in the worst case) at least doubly exponential in the number of variables of the input system. Therefore, the worst case complexity of the generalized Fourier-Motzkin algorithm of Section 4.2.1 is also at least doubly exponential in the number of variables. Experiments conducted with the loop parallelizer *LooPo* showed that usually problems with constant coefficients and 6 to 8 variables can be solved in reasonable time on current desktop computers with Fourier-Motzkin, but we encounter problems with more than 10 variables. Therefore, we have been looking for an alternative algorithm which produces output equivalent to Fourier-Motzkin elimination and which also works in the presence of non-linear parameters.

Given a polyhedron P in the variables $x_1, \ldots, x_n$, Fourier-Motzkin successively calculates projections of P to the dimensions $x_1, \ldots, x_i$, for $i \in \{1, \ldots, n\}$, and (at the same time) finds the lower and upper bounds for x_i in terms of $x_1, \ldots, x_{i-1}$.

Our approach is to calculate instead the lower and upper bounds for the dimensions $x_1, \ldots, x_n$ using quantifier elimination (without explicit projections). We have to solve two main problems to achieve this:

(1) We have to find a logical formula with describes the property "… is a lower (upper) bound" of a certain dimension.

(2) Since we are working with inputs containing non-linear parameters, we cannot simply use all lower and upper bounds together in a single solution, but we have to arrange appropriate conditionals for the different possible cases and output only the respective lower and upper bounds in each of the cases. In addition, we have to look for the condition under which the polyhedron does not contain a lower/upper bound for a given variable.

We have tried several different ways to express (1) and (2) as quantifier elimination problems. Unfortunately, we discovered that most of them do not work satisfactorily. The main difficulties are to avoid the generation of infinitesimals (cf. Section 3.2.6) in the answers and to minimize the number of cases produced by the algorithm. We have found an algorithm which works but fails to outperform Fourier-Motzkin in practice (see Chapter 5). We present a sketch of this algorithm in the following:

Step (1) Let φ be a formula describing the polyhedron P. We note that $\mathit{frvar}(\varphi) = \{x_1,\dots,x_n\} \cup \{p_1,\dots,p_m\}$, where $p_1,\dots,p_m$ are the (possibly non-linear) parameters of the polyhedron P. To find the lower and upper bounds for a variable x_i, we look for the minimum or maximum of x_i in P in dependence of $x_1,\dots,x_{i-1}$.

Given some fresh variable y, the formula

$$\psi := \exists x_{i+1} \cdots \exists x_n \big(\varphi \wedge \forall y \forall x_{i+1} \cdots \forall x_n (\varphi[x_i/y] \to x_i \le y)\big)$$

expresses that, for given values of $p_1,\dots,p_m$ and $x_1,\dots,x_{i-1}$, there exists a point $p \in P$ with $p = (x_1,\dots,x_{i-1},x_i,\dots)$, and x_i is minimal for the given choice of $x_1,\dots,x_{i-1}$.

If we now submit the formula $\exists x_i\, \psi$ to a quantifier elimination with answer system and demand answers for x_i, we will get a list

$$\big\{(\delta_j, \{x_i = t_j\}) \mid j \in \{1,\dots,k\}\big\} \tag{4.2}$$

for some $k \in \mathbb{N}$, conditions $\delta_j \in \mathit{Qf}(\{x_1,\dots,x_{i-1},p_1,\dots,p_m\},\Sigma_{ord})$, and terms $t_j \in \mathit{Tm}(\{x_1,\dots,x_{i-1},p_1,\dots,p_m\},\Sigma_{ord'})$.

When a condition δ_j holds for given values of $p_1,\dots,p_m$ and $x_1,\dots,x_{i-1}$, then t_j is the minimum of x_i in P (in dependence of $x_1,\dots,x_{i-1}$ and $p_1,\dots,p_m$). Since the formula ψ is linear in $\{x_1,\dots,x_n,y\}$, Lemma 3.17 implies that the term t_j is linear in $\{x_1,\dots,x_{i-1}\}$.

Step (2) To construct an appropriate case distinction for our result, we now ask the question, when (i.e., under which condition depending on the parameters) a given t_j is a lower bound for x_i. Note that the bounds t_j from above are guarded with conditions δ_j, which are in the parameters and the variables $x_1,\dots,x_{i-1}$ and express that t_j is the minimum of x_i. We are now looking for the conditions γ_j in the parameters only, which describe that t_j is a lower bound, possibly not a sharp bound, for *every* $x_1,\dots,x_{i-1}$.

In quantifier elimination, this question is expressed by the following formula

$$\mu_j := \mathit{denom}(t_j) \ne 0 \wedge \forall x_1 \cdots \forall x_n (\varphi \to \beta_j) \tag{4.3}$$

where $\beta_j := (x_i \ge y)[y /\!/ t_j]$ (for some fresh $y \in \mathcal{V}$) and

$$\mathit{denom}(t) := \begin{cases} s_2 & \text{if } t \text{ is of the form } s_1 \cdot \overline{s_2^{-1}} \\ 1 & \text{otherwise} \end{cases}$$

expresses that—for given values of the parameters—t_j is defined (since its denominator is not zero) and that t_j is a lower bound for x_i. The formula β_j is constructed form t_j using virtual substitution, since t_j is a $\Sigma_{ord'}$-term, but μ_j must be a Σ_{ord}-formula.

Calculating quantifier-free equivalents of the μ_j, we get formulas λ_j which are in the parameters only, and each λ_j is equivalent to t_j being a lower bound for x_i. We now get a set of condition/bound pairs

$$B = \big\{(\lambda_j, t_j) \mid j \in \{1,\dots,k\}\big\}$$

where t_j is a (probably not sharp) lower bound if and only if λ_j holds. Our implementation tries to minimize the number of elements in B by checking whether $\forall(\lambda_j \to \lambda_{j'})$ holds for some $j \neq j'$: in this case, $t_{j'}$ is dominated by t_j and we can remove $(\lambda_{j'}, t_{j'})$ from B. We construct a condition/set-of-bounds set where each condition guards exactly the terms which are bounds under that condition:

$$L := \Big\{ \big(\bigwedge_{j\in J} \lambda_j \wedge \bigwedge_{j\notin J} \neg\lambda_j, \{t_j \mid j \in J\}\big) \;\big|\; \varnothing \neq J \subseteq \{1,\ldots,k\} \Big\}$$

In other words, we construct a complete case distinction on which λ_j hold and which do not hold. Using the technique outlined in Section 4.1.2, we construct a decision tree from L whose leaves are the lower bounds for x_i (guarded with conditions in the parameters at the inner nodes of the tree). This takes care of the case that the polyhedron has a lower bound for x_i, but we have to consider the case that the polyhedron is unbounded. The formula ν (with a fresh variable y) expresses that the polyhedron is non-empty and has no lower bound for x_i:

$$\nu := \exists x_1 \cdots \exists x_n \big(\varphi \wedge \forall y(\varphi[x_i/y] \to \varphi[x_i/y-1])\big)$$

The quantifier-free equivalent is a formula in the parameters, and this condition can be added to the case distinction for the additional case "no lower bound for x_i."

Constructing the solution We can repeat Steps (1) and (2) for every x_i with $i \in \{1,\ldots,n\}$ and also for the upper bounds instead of the lower bounds (only the orientation of some relation symbols has to be changed, and the -1 in the formula for unboundedness has to be changed to $+1$). This yields decision trees for the lower and upper bounds of every x_i. Combining these trees into one big tree by "appending" one tree to all the leaves of another tree repeatedly for all the trees and collecting the sets of lower and upper bounds at the final leaves (e.g., by repeatedly using the *treeMap* combinator from Section 4.2), we finally get a decision tree carrying appropriate lower and upper bounds for each variable at its leaves. This result is equivalent to applying the generalized Fourier-Motzkin algorithm to the given inequality system.

This algorithm is largely obvious, but there is a potential problem hidden in it. Let us look at Figure 4.3 to illustrate this. The only necessary lower bound for x_2 in the depicted triangle is $x_2 \geq 1$, so we would like the Set (4.2) to be

$$\{(1 \leq x_1 \wedge x_1 \leq 7, x_2 = 1)\}$$

in this case. Unfortunately, another correct quantifier elimination result is

$$\begin{aligned}\{(1 \leq x_1 \wedge x_1 \leq 7 \wedge x_1 \neq 4,\ & x_2 = 1),\\ &(x_1 = 4,\ x_2 = x_1 - 3)\}\end{aligned}$$

Obviously, the hyperplane $x_2 = x_1 - 3$ (depicted as a dashed line in Figure 4.3) is *not* suitable for a sorted description of the triangle. The problem which arises from this is not that such unsuitable bounds are mixed among the desired bounds: the unsuitable bounds will be filtered by Formula (4.3), since μ_j is equivalent to false for such bounds. Theoretically, a necessary bound (in our example $x_2 = 1$) could be missing from the computed answer, because all the points on it are covered by unsuitable bounds.

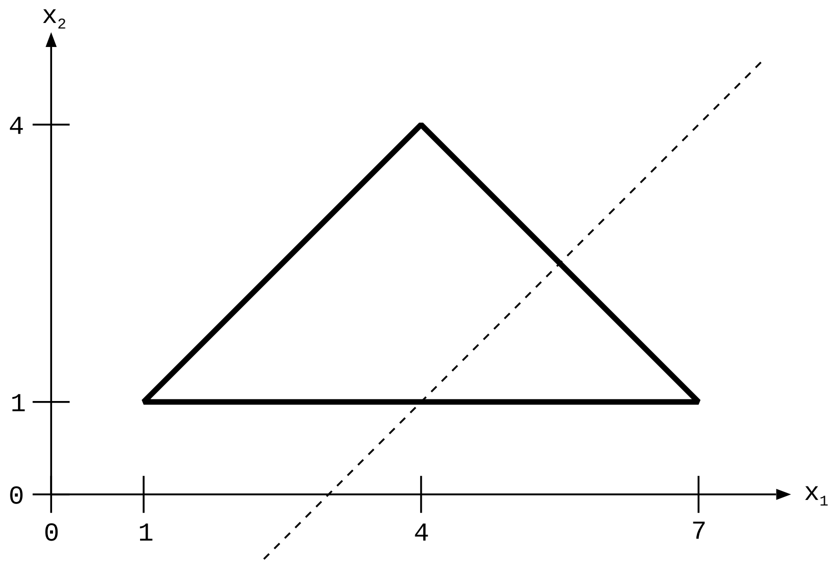


Figure 4.3: An undesired lower bound for x_2

We have not developed a formal proof showing that the hypothetical situation just described can never happen. The reason is that, as our experiments in Chapter 5 show, we do not achieve better performance than Fourier-Motzkin elimination with this approach (and it often fails due to memory exhaustion), although we spent quite some time experimenting with different variants. We just present a conjecture stating formally that Set (4.2) contains all the necessary lower bounds (upper bounds are similar), i.e., terms which have the geometric properties required for a sorted description of the input polyhedron. We also give our ideas for the reasoning in a formal proof of the conjecture.

Conjecture 4.1 *Let P be a polyhedron in the variables $x_1,\dots,x_n$ and parameters $p_1,\dots,p_m$, and let P_i be the projection of P to the dimensions $x_1,\dots,x_i$. Let $(y_1,\dots,y_i)$ be a point of P_i such that y_i is minimal with respect to $(y_1,\dots,y_{i-1})$. Then there exists a term t in Set* (4.2) *such that the hyperplane h defined by $x_i = t$ and the halfspace H defined by $x_i \geq t$ have the following properties:*

(a) $(y_1,\dots,y_i) \in h$

(b) $P_i \subseteq H$

Proof idea. Let P_i be the projection of P to the dimensions $x_1,\dots,x_i$. For any point $(y_1,\dots,y_i) \in P_i$, where y_i is minimal with respect to $y_1,\dots,y_{i-1}$, there exists a halfspace G defined by the inequality $x_i \geq e$ (for a term e which is linear in $x_1,\dots,x_{i-1}$) with associated hyperplane g which satisfies the following conditions: $P_i \subseteq G$, $(y_1,\dots,y_i) \in g$, $b := P_i \cap g$ has maximal dimensionality for all possible G. Since b cannot be represented as a union of finitely many sets whose dimensionalities are lower than the dimensionality of b, and every point in b satisfies an equation $x_i = t_j$ for some $j \in \{1,\dots,k\}$ (every point from b is lexicographically minimal in P_i by definition), there exists an $l \in \{1,\dots,k\}$ in Set (4.2) such that b is a subset of the hyperplane h defined by $x_i = t_l$. The hyperplane h and the halfspace H defined by $x_i \geq t_l$ have the properties stated in the conjecture. $\square$

4.3.3 Convex and Disjoint Unions of Polyhedra

Beside calculating a sorted description of polyhedra, other algorithms used in the polyhedron model are the calculation of convex unions and disjoint unions of polyhedra. Convex unions are often used to calculate an "approximation" of a set of polyhedra through a single polyhedron which is a superset of the individual polyhedra.

Definition 4.2 Given the finitely many polyhedra $P_1, \ldots, P_n$, the *convex union* of these polyhedra is a polyhedron P which satisfies the following two conditions:

(1) $P_i \subseteq P$ for every $i \in \{1, \ldots, n\}$,

(2) No polyhedron $Q \subsetneqq P$ satisfies (1), i.e., P is the smallest polyhedron satisfying (1) (with regard to set inclusion).

A disjoint union of polyhedra is a representation of the union of given polyhedra through pairwise disjoint polyhedra.

Definition 4.3 Given finitely many polyhedra $P_1, \ldots, P_n$, a *disjoint union* of these polyhedra is a finite set of polyhedra $Q_1, \ldots, Q_k$ which satisfies the following two conditions:

(1) $P_1 \cup \cdots \cup P_n = Q_1 \cup \cdots \cup Q_k$,

(2) $Q_1, \ldots, Q_k$ are pairwise disjoint, i.e., $Q_i \cap Q_j = \varnothing$ for $1 \leq i < j \leq k$.

Disjoint unions of polyhedra are needed in loop parallelization during code generation when the iteration domains of different statement are combined. The iteration domain of each statement is described by a polyhedron. The polyhedra need not be disjoint, i.e., different statements can share points in the iteration domain. Usually it is desired to enumerate every point of the iteration domain only once (for all the statements). This can be achieved by two methods. The simple method is to calculate a convex union of the given iteration domains (or even a "bigger" superset of them, e.g., a rectangular superset) and to enumerate every point of this superset. It is then necessary—for every point which is enumerated—to test which of the given polyhedra the point is a member of. The more complex method is to calculate a disjoint union of the given polyhedra and to enumerate the polyhedra resulting from that calculation. This ensures that exactly the points in the given iteration domains are enumerated, and it can be statically determined which polyhedron enumerates points for which statement.

Although the use of a disjoint union ensures that no superfluous points are enumerated, it can still be desirable to use a convex union (or other, simply described superset) since the enumeration of a disjoint union can be much more complex, so that enumerating some superfluous points from the chosen superset can still be more efficient.

Convex Unions

We give here the definition of convexity and a well-known theorem about polyhedra:

Definition 4.4 A set C is called *convex*, if for every $a, b \in C$ and every $r \in [0,1]$ it follows that $r \cdot a + (1-r) \cdot b \in C$.

Theorem 4.5 *Let $P_1,\dots,P_k$ be polyhedra. Then the set*

$$C := \{\sum_{i=1}^{k} r_i \cdot p_i \mid p_1 \in P_1,\dots,p_k \in P_k, r_1,\dots,r_k \geq 0, \sum_{i=1}^{k} r_i = 1\}$$

is the convex union of $P_1,\dots,P_k$ and C is a polyhedron.

This theorem yields a method to compute the convex union of given polyhedra. Let $P_1,\dots,P_k$ be (parametric) polyhedra defined by the formulas $\psi_1,\dots,\psi_k$. We assume that the free variables of $\psi_1,\dots,\psi_k$ are $\{x_1,\dots,x_n,p_1,\dots,p_m\}$, where $x_1,\dots,x_n$ denote the n dimensions of the polyhedra and $p_1,\dots,p_m$ are the parameters the polyhedra are parametrized with. Then the following formula φ describes the convex union of $P_1,\dots,P_k$:

$$\varphi := \exists r_1 \cdots \exists r_k \quad \exists x_{1,1} \cdots \exists x_{1,n} \cdots \exists x_{k,1} \cdots \exists x_{k,n}$$
$$\Big(\bigwedge_{i=1}^{k} \psi_i[x_1/x_{i,1},\dots,x_n/x_{i,n}] \wedge \bigwedge_{j=1}^{n} \sum_{i=1}^{k} (r_i \cdot x_{i,j}) = x_j \wedge \bigwedge_{i=1}^{k} r_i \geq 0 \wedge \sum_{i=1}^{k} r_i = 1 \Big)$$

This works as follows: the formula φ claims

- the existence of a point $p_i = (x_{i,1},\dots,x_{i,n})$ in the polyhedron P_i (since $\psi_i[x_1/x_{i,1},\dots,x_n/x_{i,n}]$ holds) for every $i \in \{1,\dots,k\}$,
- the existence of coefficients $r_1,\dots,r_k \geq 0$ with $\sum_{i=1}^{k} r_i = 1$, and
- that the point $p = (x_1,\dots,x_n)$ is a $(r_1,\dots,r_k)$-combination of the points p_i: $p = \sum_{i=1}^{k} r_i \cdot p_i$.

Theorem 4.5 shows that this claim is true (for given values of the parameters $p_1,\dots,p_m$) if and only if the point $(x_1,\dots,x_n)$ lies in the convex union of $P_1,\dots,P_k$. Theorem 4.5 also states that φ describes a polyhedron. Hence, φ describes the convex union of $P_1,\dots,P_k$ in the variables $x_1,\dots,x_n$ and the parameters $p_1,\dots,p_m$.

Therefore, it is possible to feed the formula φ (or its quantifier-free equivalent) into the sorting algorithm of Section 4.3.2 to get a description of the convex union in terms of a sorted inequality system.

Disjoint Unions

We have calculated convex unions by describing the desired result with a first-order formula and, since the convex union is again a polyhedron, used a sorting algorithm to get a description of that polyhedron in an adequate form. The situation is different for disjoint unions, since the desired solution consists of an unknown number of (disjoint) polyhedra.

To find a description of the disjoint union we use an approach that manipulates a quantifier-free formula and extracts descriptions of the individual disjoint polyhedra from it. The main transformation step is the conversion of a formula into disjunctive normal form.

Lemma 4.6 *For every positive formula $\psi \in Qf(\mathcal{V},\Sigma)$ there exist $n \in \mathbb{N}$, $k_1,\dots,k_n \in \mathbb{N}$ and atomic formulas $a_{i,j} \in At(\mathcal{V},\Sigma)$ $(i \in \{1,\dots,n\},\ j \in \{1,\dots,k_i\})$ such that*

$$\models \Big(\psi \leftrightarrow \bigvee_{i=1}^{n} \bigwedge_{j=1}^{k_i} a_{i,j}\Big)$$

$\bigvee_{i=1}^{n} \bigwedge_{j=1}^{k_i} a_{i,j}$ *is called a* disjunctive normal form *of* ψ.

Proof. Let ψ be a positive quantifier-free formula. We show the existence of a positive formula which is equivalent to ψ and in disjunctive normal form by induction on the structure of ψ.

$\psi \in At(\mathcal{V},\Sigma)$: ψ is in disjunctive normal form by definition.

$\psi = (\psi_1 \vee \psi_2)$: By the induction hypothesis, there exist positive formulas φ_1 and φ_2 which are in a disjunctive normal form and logically equivalent to ψ_1 and ψ_2, respectively. Then $\varphi_1 \vee \varphi_2$ is in disjunctive normal form, positive, and logically equivalent to ψ.

$\psi = (\psi_1 \wedge \psi_2)$: By the induction hypothesis, ψ_1 is equivalent to $\bigvee_{i=1}^{n} \varphi_i$ and ψ_2 is equivalent to $\bigvee_{j=1}^{m} \varphi'_j$, for some $m,n \geq 1$, and $\varphi_1,\ldots,\varphi_n,\varphi'_1,\ldots,\varphi'_m$ are conjunctions of atomic formulas. ψ is equivalent to $\bigvee_{i=1}^{n} \varphi_i \wedge \bigvee_{i=1}^{m} \varphi'_i$, which is equivalent to $\bigvee_{i=1}^{n}(\varphi_i \wedge \bigvee_{j=1}^{m} \varphi'_j)$. This formula is in turn equivalent to $\bigvee_{i=1}^{n}\big(\bigvee_{j=1}^{m}(\varphi_i \wedge \varphi'_j)\big)$, and this is a positive formula in disjunctive normal form.

□

Since, in the structure $\underline{\mathbb{R}}$, every quantifier-free formula is equivalent to a positive formula (cf. Lemma 3.7), we have the following corollary:

Corollary 4.7 *In the structure $\underline{\mathbb{R}}$, every quantifier-free formula $\psi \in Qf(\mathcal{V},\Sigma_{ord})$ is equivalent to a positive formula φ which is in disjunctive normal form.*

Using this corollary we can construct a disjunctive normal form $\bigvee_{i=1}^{l} \gamma_i$ (for some $l \in \mathbb{N}$) of a formula φ such that

$$\begin{aligned} &\underline{\mathbb{R}} \models \varphi \leftrightarrow \bigvee_{i=1}^{l} \gamma_i \\ &\underline{\mathbb{R}} \models \neg(\gamma_i \wedge \gamma_j) \quad \text{for all } i,j \in \{1,\ldots,l\},\ i \neq j \end{aligned} \tag{4.4}$$

i.e., the sets defined by the γ_i are pairwise disjoint. The following algorithm, which we call *disj*, computes a set of formulas $\{\gamma_1,\ldots,\gamma_l\}$ with the properties stated in the Specification (4.4):

Algorithm *disj*

1. Calculate a positive disjunctive normal form of φ, i.e., find (for some $k \in \mathbb{N}$) formulas $\psi_1,\ldots,\psi_k$ which are conjunctions of atomic formulas such that $\underline{\mathbb{R}} \models \varphi \leftrightarrow \bigvee_{i=1}^{k} \psi_i$. To ensure termination, the positive disjunctive normal form must satisfy the following two conditions:

   (p1) Every term appearing in the formula $\bigvee_{i=1}^{k} \psi_i$ also appears in φ either as-is or negated.

   (p2) ψ_1 in $\bigvee_{i=1}^{k} \psi_i$ is feasible, i.e., there exists an environment $h \in \mathbb{R}^{\mathcal{V}}$ such that $\underline{\mathbb{R}} \models_h \psi_1$.

2. If $k = 0$ we define

$$disj(\varphi) := \varnothing$$

   to be the result of the algorithm *disj* in this case.

3. Otherwise, construct the formulas $\delta_1, \ldots, \delta_k$ from $\psi_1, \ldots, \psi_k$ according to the following schema:

$$\begin{aligned}
\delta_1 &:= \psi_1 \\
\delta_2 &:= \psi_2 \wedge \neg\psi_1 \\
\delta_3 &:= \psi_3 \wedge \neg\psi_1 \wedge \neg\psi_2 \\
&\vdots \\
\delta_k &:= \psi_k \wedge \neg\psi_1 \wedge \cdots \wedge \neg\psi_{k-1}
\end{aligned}$$

4. The result of $disj(\varphi)$ is computed as follows:

$$disj(\varphi) := \{\delta_1\} \cup \bigcup_{i=2}^{k} disj(\delta_i)$$

This means that δ_1 is directly part of the result and the recursive application of the algorithm yields appropriate formulas (i.e., conjunctions of atomic formulas) which describe $\delta_2, \ldots, \delta_k$ disjointly.

As an example for the application of the algorithm *disj*, let us look at the set depicted in Figure 4.4 (a). It is a union of two rectangles defined by $\psi_1 = (x \geq 1 \wedge x \leq 4 \wedge y \geq 1 \wedge y \leq 3)$ and $\psi_2 = (x \geq 2 \wedge x \leq 5 \wedge y \geq 2 \wedge y \leq 5)$. If we now compute $disj(\psi_1 \vee \psi_2)$, a possible disjunctive normal form of $\psi_1 \vee \psi_2$, is, of course, the formula $\psi_1 \vee \psi_2$ itself, and this disjunctive normal form satisfies both properties (p1) and (p2). The algorithm constructs the two formulas $\delta_1 = \psi_1$ and $\delta_2 = \psi_2 \wedge \neg\psi_1$; the disjoint sets defined by δ_1 and δ_2 are shown in Figure 4.4 (b). Figure 4.4 (c) and (d) illustrate the recursive application of *disj* to the formula δ_2. A possible positive disjunctive normal form for δ_2 is $\psi'_1 \vee \psi'_2$ with $\psi'_1 = (x > 4 \wedge x \leq 5 \wedge y \geq 2 \wedge y \leq 5)$ and $\psi'_2 = (x \geq 2 \wedge x \leq 5 \wedge y > 3 \wedge y \leq 5)$. This gives again rise to two δ-formulas, $\delta'_1 = \psi'_1$ and $\delta'_2 = \psi'_2 \wedge \neg\psi'_1$. A positive disjunctive normal form for δ'_2 is $x \geq 2 \wedge x \leq 4 \wedge y > 3 \wedge y \leq 5$, which consists of one disjunct only. The recursion then stops and a disjoint union of the set defined by $\psi_1 \vee \psi_2$ is given by $\{\delta_1, \delta'_1, \delta'_2\}$.

The termination of *disj* depends on the properties of the algorithm used to compute the positive disjunctive normal form of φ in Step 1 of the algorithm. It is possible that *disj* never terminates if "unfortunate" disjunctive normal forms are computed. For example, if we start with the one-dimensional polyhedron $[0,4]$ described by the formula $\varphi = (x \geq 0 \wedge x \leq 4)$, a valid disjunctive normal form is $(x \geq 0 \wedge x < 2) \vee (x \geq 2 \wedge x \leq 4)$ which leads to the two δ-formulas $\delta_1 = (x \geq 0 \wedge x < 2)$ and $\delta_2 = \big((x \geq 2 \wedge x \leq 4) \wedge \neg(x \geq 0 \wedge x < 2)\big)$. δ_2 simplifies to $x \geq 2 \wedge x \leq 4$ and a valid disjunctive normal form for this formula is $(x \geq 2 \wedge x < 3) \vee (x \geq 3 \wedge x \leq 4)$. This, again, leads to two δ-formulas and the process of dividing δ_2 into two small intervals at each step can be repeated ad infinitum.

Such a recursion process which "invents" new sub-polyhedra does not have property (p1): The given formula $x \geq 0 \wedge x \leq 4$ has the two terms x and $x-4$ (if we write the formula in the canonical form $x \geq 0 \wedge x - 4 \leq 0$, where the right side of an atomic formula is always 0), but the given disjunctive normal form contains the atomic formula $x \leq 2$ and hence the term $x-2$ (of the canonical form $x - 2 \leq 0$), which did not appear in the original formula (nor did $-(x-2)$). This invention of new terms can lead to the nontermination of *disj*. The rationale for (p2) is that an infeasible ψ_1 could always appear as the first disjunct of the

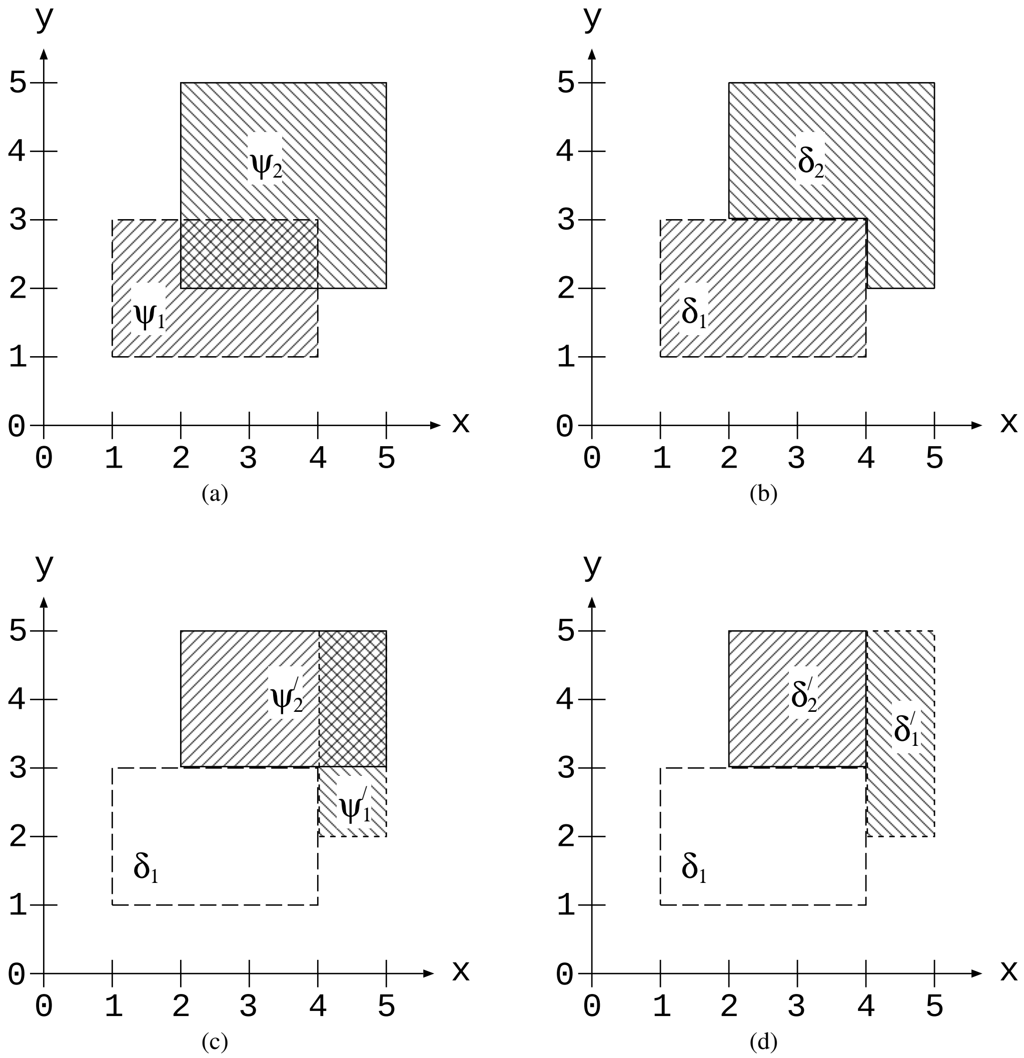


Figure 4.4: Algorithm *disj* in an example

normal form so that—even if φ is infeasible—the case $k = 0$ might never occur during the recursion.

The existence of an algorithm to compute a positive disjunctive normal form of any given quantifier-free formula φ in $\mathbb{R}$ which satisfies (p1) is shown by the proofs of Lemma 3.7 and Lemma 4.6. The transformation rules given there transform φ into a positive disjunctive normal form without using terms not appearing in φ (except for negations of given terms). To satisfy (p2), one can test the feasibility of every ψ_i (e.g., by applying quantifier elimination to the formula $\exists(\psi_i)$). Only the ψ_i with $\mathbb{R} \vDash \exists(\psi_i)$ are retained in the disjunctive normal form and the others are dropped.

Theorem 4.8 *Let $\varphi \in Qf(\mathcal{V}, \Sigma_{ord})$. Then $disj(\varphi)$ terminates if properties (p1) and (p2) are satisfied by the algorithm computing the positive disjunctive normal form in Step 1 of algorithm disj, and the result is a set*

$$disj(\varphi) = \{\gamma_1, \ldots, \gamma_l\}$$

for some $l \in \mathbb{N}$, $\gamma_1, \ldots, \gamma_l \in Qf(\mathcal{V}, \Sigma_{ord})$ *such that the conditions*

$$\underline{\mathbb{R}} \models \varphi \leftrightarrow \bigvee_{i=1}^{l} \gamma_i$$

$$\underline{\mathbb{R}} \models \neg(\gamma_i \wedge \gamma_j) \quad \textit{for all } i, j \in \{1, \ldots, l\},\ i \neq j$$

from Specification (4.4) *hold and the* γ_i *are conjunctions of atomic formulas.*

Proof. First, we prove inductively the correctness of the algorithm *disj* with regard to the given specification:

(a) By construction, $\bigvee_{i=1}^{k} \psi_i$ is equivalent to φ (in $\underline{\mathbb{R}}$). In the case $k = 0$ (which means that φ is infeasible), the algorithm returns $\varnothing$ and since the empty disjunction is logically equivalent to “false”, the algorithm returns a correct result.

(b) For the case $k \geq 1$, we prove that $\underline{\mathbb{R}} \models \bigvee_{i=1}^{k} \psi_i \leftrightarrow \bigvee_{i=1}^{k} \delta_i$. Let $h \in \mathbb{R}^{\mathcal{V}}$ be an environment. Assume that $\underline{\mathbb{R}} \models_h \bigvee_{i=1}^{k} \psi_i$. Then there is at least one $i \in \{1, \ldots, k\}$, such that $\underline{\mathbb{R}} \models_h \psi_i$. Let $j \in \{1, \ldots, k\}$ be minimal with $\underline{\mathbb{R}} \models_h \psi_j$. Then, for $i \in \{1, \ldots, j-1\}$, we have $\underline{\mathbb{R}} \models_h \neg\psi_i$ and, therefore, $\underline{\mathbb{R}} \models_h \delta_j$, since $\delta_j = \psi_j \wedge \bigwedge_{i=1}^{j-1} \neg\psi_i$. This shows that $\underline{\mathbb{R}} \models_h \bigvee_{i=1}^{k} \delta_i$.

Conversely, assume $\underline{\mathbb{R}} \models_h \bigvee_{i=1}^{k} \delta_i$. Then there exists some $j \in \{1, \ldots, k\}$ with $\underline{\mathbb{R}} \models_h \delta_j$. Since $\delta_j = \psi_j \wedge \bigwedge_{i=1}^{j-1} \neg\psi_i$ this implies $\underline{\mathbb{R}} \models_h \psi_j$ and hence $\underline{\mathbb{R}} \models_h \bigvee_{i=1}^{k} \psi_i$.

(c) The sets defined by $\delta_1, \ldots, \delta_k$ are pairwise disjoint, since for any $1 \leq l < m \leq k$, we have $\delta_l \wedge \delta_m = \left(\psi_l \wedge \bigwedge_{i=1}^{l-1} \neg\psi_i\right) \wedge \left(\psi_m \bigwedge_{i=1}^{m-1} \neg\psi_i\right)$, but this conjunction contains both ψ_l and $\neg\psi_l$ and can never be true.

(d) By hypothesis, the recursive application of *disj* calculates, for $i \in \{2, \ldots, k\}$, sets $disj(\delta_i)$ such that $\underline{\mathbb{R}} \models \bigvee disj(\delta_i) \leftrightarrow \delta_i$, and the sets $disj(\delta_i)$ consist of formulas which are conjunctions of atomic formulas defining pairwise disjoint sets. Together with (b) and (c) this implies that

$$\underline{\mathbb{R}} \models \varphi \leftrightarrow \delta_1 \vee \bigvee_{i=2}^{k} \left(\bigvee disj(\delta_i)\right)$$

and all the formulas in $\{\delta_1\} \cup \bigcup_{i=2}^{k} disj(\delta_i)$ are conjunctions of atomic formulas which describe pairwise disjoint sets.

The following part of the proof establishes that *disj* eventually terminates.

We define the following notation: for a given quantifier-free formula σ, let S be the set defined by σ. Define

$$\mathcal{S}(\sigma) := \{A \mid A \subseteq S \text{ and } A \text{ can be defined by a quantifier-free formula } \tau \\ \text{whose terms appear negated or unnegated in } \sigma\}$$

Since there is only a finite number of terms in σ, the number of atomic formulas which can be constructed from these terms (and their negations) is also finite. This implies that the set $\mathcal{S}(\sigma)$ is finite.

If the above properties (p1) and (p2) hold, we can argue as follows that $|\mathcal{S}(\varphi)|$ is a termination function. Due to property (p1) each of the formulas $\delta_1,\dots,\delta_k$ defines a set from $\mathcal{S}(\varphi)$. Property (p2) ensures that δ_1 $(=\psi_1)$ defines a set $D \neq \varnothing$. The construction of $\delta_2,\dots,\delta_k$ now makes sure that every δ_i for $i \in \{2,\dots,k\}$ defines a set from $\mathcal{S}(\varphi)\setminus\{D\}$. This means that $\mathcal{S}(\delta_i) \subsetneqq \mathcal{S}(\varphi)$ and hence $|\mathcal{S}(\delta_i)| < |\mathcal{S}(\varphi)|$. □

Remarks:

- *disj* is easily used to compute a disjoint union of (parametric) polyhedra. Let $\varphi_1,\dots\varphi_r$ be formulas describing (parametric) polyhedra. Obviously $\varphi := \varphi_1 \vee \dots \vee \varphi_r$ describes the union of these polyhedra and, hence, $\mathit{disj}(\varphi)$ represents a disjoint union of the polyhedra described by $\varphi_1,\dots,\varphi_r$.

- Through the negations involved in the construction of the δ_i formulas, $\mathit{disj}(\varphi)$ can contain strict inequalities, even if φ only contains weak inequalities. We can transform the strict inequalities into weak ones, if we know that the parameters are integral and we are only interested in the integral points in the disjoint union (cf. Section 2.1.3).

- Looking more closely at the proof, it is clear that (p2) ensures that every $\gamma \in \mathit{disj}(\varphi)$ is feasible. This does not mean that every $\gamma \in \mathit{disj}(\varphi)$ represents a non-empty polyhedron. We have to remember that we did not make any distinctions between variables and parameters in the algorithm *disj*: it treats the parameters like variables. What (p2) ensures is that for every $\gamma \in \mathit{disj}(\varphi)$ there are values for the variables and the parameters which make γ true, but there can be values for the parameters such that no values for the variables make γ true. If we want to ensure that we only enumerate non-empty polyhedra (if we produce code from the result calculated by *disj*), we can compute for every $\gamma \in \mathit{disj}(\varphi)$ a quantifier-free equivalent of

$$\exists x_1 \cdots \exists x_n\,(\gamma)$$

  which is a condition in the parameters and expresses that γ represents a non-empty polyhedron. We can then make case distinctions on these conditions and form a decision tree carrying only the non-empty polyhedra of the disjoint union at its leaves.

Chapter 5

Sample Applications

In this chapter, we discuss briefly some experiments we conducted with the implementations of the algorithms described in the previous sections. We cannot show all the intermediate formulas and the final results here, since even simple inputs often yield very big outputs (compared to the input).

We ran the experiments on an AMD Athlon™ XP 1700+ Processor with 1467 MHz and 512 MB RAM. The software we used is REDUCE 3.7 and the Glasgow Haskell Compiler (GHC) Version 5.04.1.

5.1 Convex and Disjoint Union

5.1.1 Convex Union

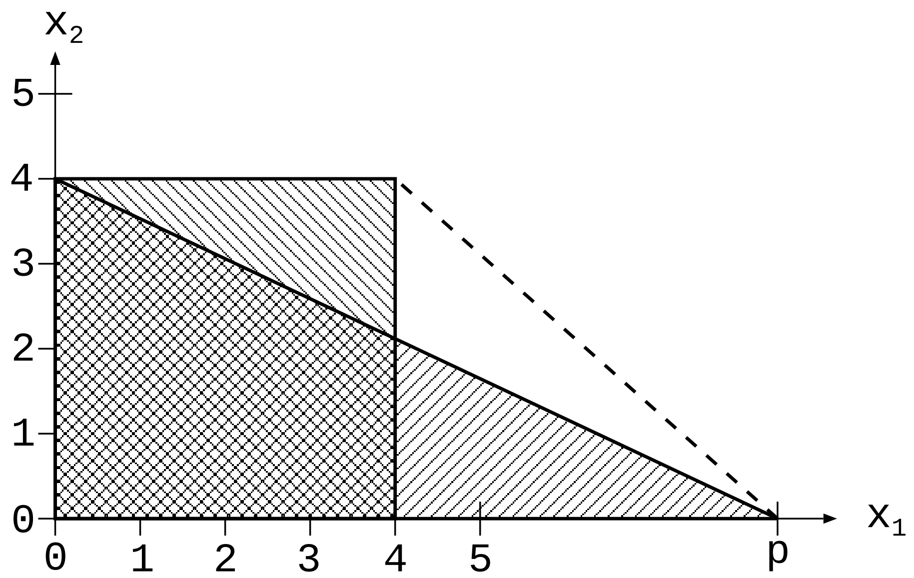


Figure 5.1: Convex union example

As an example of the convex union of polyhedra, let us look at the two polyhedra shown in Figure 5.1. We are looking for the convex union of the square with corners $(0,0)$, $(4,0)$,

$(0,4)$, $(4,4)$ and the triangle with corners $(0,0)$, $(0,4)$, $(p,0)$ for $p > 0$. The formulas defining these polyhedra are

$$\sigma := 0 \le x_1 \wedge 0 \le x_2 \wedge x_1 \le 4 \wedge x_2 \le 4$$

for the square and

$$\tau := 0 \le x_1 \wedge 0 \le x_2 \wedge p \cdot x_2 \le -4 \cdot x_1 + 4 \cdot p$$

for the triangle. Figure 5.1 shows the missing part to make the union of the square and the triangle convex (in the case $p > 4$), indicated by a dashed line. Using the method outlined in Section 4.3.3, we can construct a formula φ expressing the convex union. Then, we calculate a quantifier-free equivalent ψ of φ. We do this by first applying REDLOG's quantifier elimination method, followed by a degree decreaser (to replace some atomic formulas with quadratic terms by linear formulas), and finally a formula simplifier (a so-called Groebner simplifier). The resulting formula ψ consists of 149 atomic formulas. The sorting algorithm (Section 4.3.2) produces a decision tree with 11 leaves. Unfortunately, the case distinction contained in this tree has a total of 204 atomic formulas and the application of REDLOG's formula simplifiers did not reduce this number considerably.

The total computation time was 96 seconds.

5.1.2 Disjoint Union

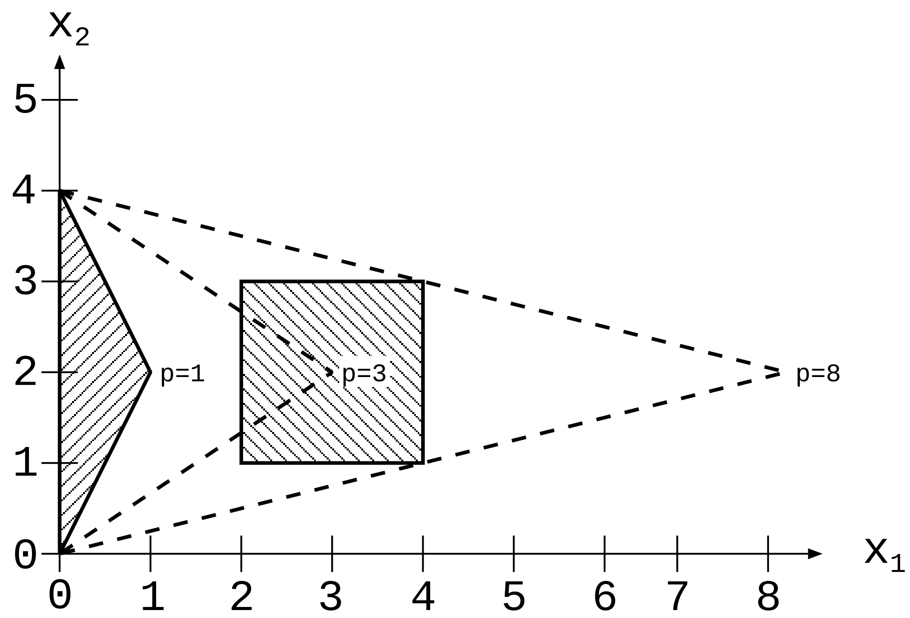


Figure 5.2: Disjoint union example

For the disjoint union example, we use again a triangle and a square, but in different positions. As is shown in Figure 5.2, the square has its corners at $(2,1)$, $(4,1)$, $(4,3)$, and $(2,3)$. The triangle has two fixes corners at $(0,0)$ and $(0,4)$ and a variable corner at $(p,2)$ for $p \ge 0$. The figure illustrates that, for $p \ge 2$, the triangle and the square share points and for $p \ge 8$ the square is a subset of the triangle.

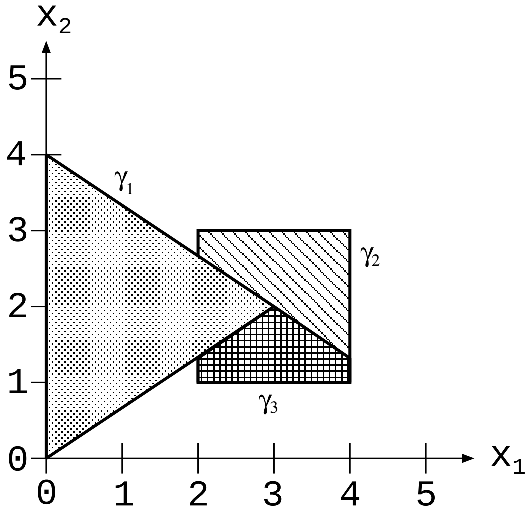


Figure 5.3: Result of algorithm *disj* for $p = 3$

The inequalities for the square are

$$\sigma := 2 \le x_1 \wedge x_1 \le 4 \wedge 1 \le x_2 \wedge x_2 \le 3$$

and for the triangle

$$\tau := 0 \le x_2 \wedge 2 \cdot x_1 \le p \cdot x_2 \wedge p \cdot x_2 \le -2 \cdot x_1 + 4 \cdot p$$

Applying the algorithm *disj* to the formula $\sigma \vee \tau$ yields (after less than one second of computation time) the result $\{\gamma_1, \gamma_2, \gamma_3\}$. The sets defined by γ_1, γ_2, γ_3 for $p = 3$ are shown in Figure 5.3. We convert the strict inequalities in the formulas γ_1, γ_2, γ_3 into systems of weak inequalities (assuming that p is integral and that we are only interested in the integral points of the disjoint union, cf. Lemma 2.6) and get three pairwise disjoint polyhedra G_1, G_2, G_3 (where G_i is derived from γ_i, for $i \in \{1,2,3\}$)):

G_1	G_2	G_3
$-2x_1 - px_2 + 4p \ge 0$	$2x_1 + px_2 - 4p - 1 \ge 0$	$-2x_1 - px_2 + 4p \ge 0$
$-2x_1 + px_2 \ge 0$	$x_1 \ge 2$	$2x_1 - px_2 - 1 \ge 0$
$x_1 \ge 0$	$x_1 \le 4$	$x_1 \ge 2$
	$x_2 \ge 1$	$x_1 \le 4$
	$x_2 \le 3$	$x_2 \ge 1$
		$x_2 \le 3$

The main part of algorithm *disj* is the computation of a disjunctive normal form of the input formula. The above example also shows that the result computed by *disj* depends on how the disjunctive normal form is computed. If we give our implementation of *disj* the formula $\tau \vee \sigma$ (instead of $\sigma \vee \tau$), the final result consists of 5 polyhedra, instead of 3. The reason is that the square σ is retained as one of the result polyhedra, and this requires that 4 polyhedra "surrounding" the square are produced. Maybe a more extensive analysis of algorithm *disj* for bigger inputs (with many more polyhedra) may reveal a heuristics for ordering the disjuncts of the computed disjunctive normal form.

5.2 Tiling an Index Space

Tiling is a well-known technique of partitioning an index space into congruent partitions. It is used to change the enumeration of an index space into an enumeration of the partitions ("tiles") and individual enumerations of the points in each partition. This technique is applied for cache optimization, to achieve parallelism in loop programs or to coarsen the granularity of program communications after space-time mapping. Since the mathematical tools used in the traditional polyhedron model only deal with linear parameters, some desirable applications could not be handled, especially, partitioning an index space with a parametric number of processors and arbitrary tile shapes is not possible. Wieninger [Wie97] computes a solution for parametric tiling with *rectangular* tiles only by hand.

5.2.1 The Principle of Tiling

We need three pieces of input to tile an index space [AI91]:

- an index space described by an inequality system S in the variables $x_1,\dots,x_n$,
- a base tile, i.e., an inequality system T describing the shape of the tiles using an inequality system in the variables $o_1,\dots,o_n$,
- n-dimensional vectors $l_1,\dots,l_n$ describing the translations between the base tile and the other tiles in n directions; the $n\times n$-matrix $L=(l_1\cdots l_n)$ is called a *lattice*.

The system T describes the shape of the tiles, whereas the lattice L describes how the tiles are placed next to each other. The lattice must match the system T so that the tiles do neither overlap nor leave holes in the index space, i.e., points which are not covered by any tile.

In [AI91] the inequality systems S may contain linear parameters. The system T must be parameter-free, since the lattice L must be a matrix with fixed numbers as entries. Using our techniques for the polyhedron model with non-linear parameters, we can now deal with lattices containing parameters and, therefore, with tile shapes T which depend on parameters.

5.2.2 Simple Tiling

The first example we present in detail is simply the tiling of a two-dimensional index space with a parametric tile size. We give the inequality systems in usual notation and in matrix notation. We choose a triangular index space with the corners $(0,0)$, $(n,0)$, and $(0,n)$:

$$\begin{aligned} 0 &\le x_1 \\ 0 &\le x_2 \\ x_1+x_2 &\le n \end{aligned} \qquad \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & -1 & n \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix} \ge 0 \tag{5.1}$$

We assume $n\ge 0$ to guarantee that the index space is not empty. As tile shape we use

$$\begin{aligned} 0 &\le o_2 \\ o_2 &\le p_2-1 \\ o_2 &\le o_1 \\ o_1 &\le o_2+p_1-1 \end{aligned} \qquad \begin{pmatrix} 0 & 1 & 0 \\ 0 & -1 & p_2-1 \\ 1 & -1 & 0 \\ -1 & 1 & p_1-1 \end{pmatrix} \cdot \begin{pmatrix} o_1 \\ o_2 \\ 1 \end{pmatrix} \ge 0 \tag{5.2}$$

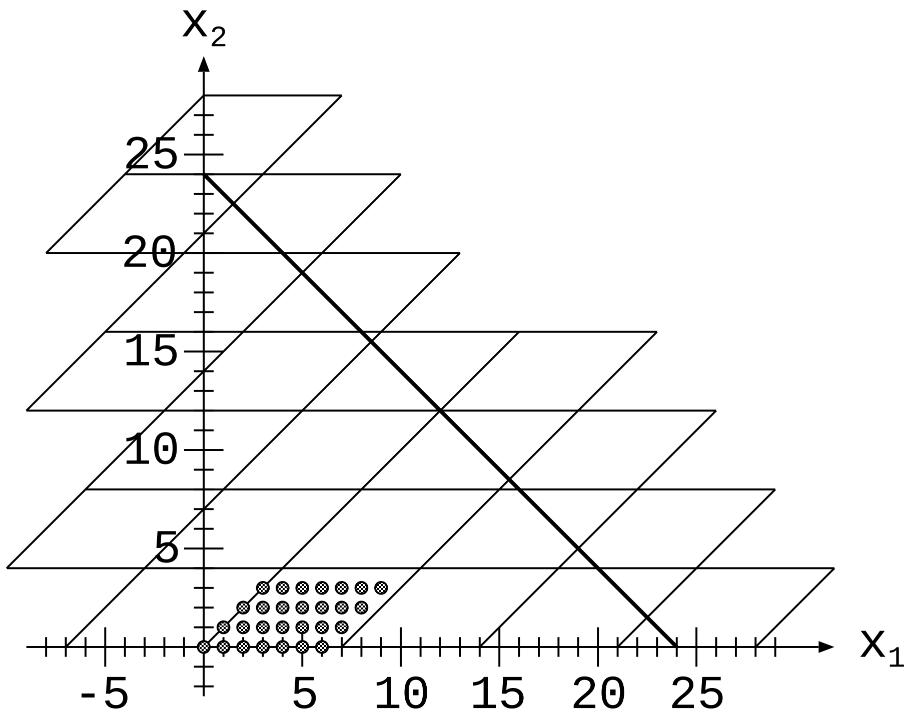


Figure 5.4: Simple tiling example for $n = 24$, $p_1 = 7$, $p_2 = 4$.

which describes a parallelogram with the corners $(0,0)$, $(p_1 - 1, 0)$, $(p_1 + p_2 - 2, p_2 - 1)$, and $(p_2 - 1, p_2 - 1)$. The parameters p_1 and p_2 are the "width" and "height" of the parallelogram. We assume $p_1, p_2 \geq 1$ to make the tiles non-empty. A lattice which matches this tile shape is

$$L = \left(\begin{array}{c|c} p_1 & p_2 \\ 0 & p_2 \end{array}\right)$$

This lattice represents a tile layout as in Figure 5.4 for $n = 24$, $p_1 = 7$, and $p_2 = 4$. The figure shows all the tiles with integral points in the triangle, and the integral points within the base tile (the tile at $(0,0)$) are shown as little circles.

A description of the tiling is the system consisting of Systems (5.1) and (5.2) together with the equation (cf. [AI91])

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = L \cdot \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} + \begin{pmatrix} o_1 \\ o_2 \end{pmatrix} \tag{5.3}$$

The complete system (consisting of Systems (5.1), (5.2), (5.3)) has 6 variables and 3 parameters. Two of the parameters occur non-linearly. We ask our implementation to sort the system such that the variables are in the order t_1, t_2, x_1, x_2

The result of the sorting is shown in Figure 5.2.2. The total computation time of applying the generalized Fourier-Motzkin to find the bounds for the dimensions t_1, t_2, x_1, x_2 and simplifying the decision tree (to yield a tree with a single leaf only) was 5 seconds when using REDLOG to simplify the decision tree and 9 seconds when using our CAD implementation. The sorting algorithm of Section 4.3.2 yielded a result after 10 seconds, but the formula

$$
\begin{array}{r|l}
 & p_1t_1 + p_2t_2 \le x_1 \\
 & p_1t_1 \le x_1 \\
\frac{-p_1-n+1}{p_1} \le t_1 & 0 \le x_1 \\
t_1 \le \frac{n}{p_1} & x_1 \le n \\
 & x_1 \le \frac{p_1}{2}t_1 + \frac{p_1+n-1}{2} \\
 & x_1 \le -p_2t_2 + n \\
 & x_1 \le p_1t_1 + p_2t_2 + p_1 \\
 & \qquad +p_2 - 2 \\
\frac{-p_1}{p_2}t_1 + \frac{-p_1-p_2+2}{p_2} \le t_2 & -p_1t_1 + x_1 - p_1 + 1 \le x_2 \\
\frac{-p_2+1}{p_2} \le t_2 & p_2t_2 \le x_2 \\
t_2 \le \frac{-p_1}{2p_2}t_1 + \frac{n}{2p_2} & 0 \le x_2 \\
t_2 \le \frac{-p_1}{p_2}t_1 + \frac{n}{p_2} & x_2 \le -p_1t_1 + x_1 \\
t_2 \le \frac{n}{p_2} & x_2 \le p_2t_2 + p_2 - 1 \\
 & x_2 \le -x_1 + n
\end{array}
$$

Figure 5.5: Sorted inequality system for the simple tiling example

simplifiers were not able to reduce the result to a decision tree with only one leaf: the result had 17 leaves and 30 atomic formulas in the conditions.

5.2.3 Tiling and Communication

As a final example, let us look at another tiling problem. The previous section shows how to enumerate the points in the index space in a tiled fashion: enumerate the tiles and, for every tile, the points within the tile. Let us now look at the communications necessary between the tiles. Let us assume that every point (x_1,x_2) in the 2-dimensional index space represents an operation which depends on some values computed by the operation at (x_1-2,x_2-1) (if (x_1-2,x_2-1) is part of the index space). The problem we want to solve is to compute which tiles have to communicate with each other. To put it in other words: the points within one tile must communicate with other points (in the same or some other tiles). We want to know between which tiles communications take place (and which points are involved). This is used in loop parallelization to avoid communications between the individual points; instead, communication is done at the tile level, i.e., every tile communicates once with its communication partners and this communication transfers all the values the points in the sending tile need to transfer to the receiving tile.

We use the same index space, tile shape, and lattice as in the previous section. The difference is that—since we are talking about sending and receiving tiles—we have to use separate variables for the sender and the receiver. We use t_1,t_2,x_1,x_2,o_1,o_2 for the sender coordinates and $t_1',t_2',x_1',x_2',o_1',o_2'$ for the receiver coordinates.

The connection between the senders and the receivers is made through two equations. Since we assume that (x_1,x_2) sends data to (x_1+2,x_2+1) (if both points are in the index space), we have the additional equations $x_1' = x_1+2$ and $x_2' = x_2+1$.

The complete inequality system describing the communications is:

$$
\begin{aligned}
&0 \le x_1 && 0 \le x'_1 \\
&0 \le x_2 && 0 \le x'_2 \\
&\quad x_1 + x_2 \le n && \quad x'_1 + x'_2 \le n \\
&0 \le o_2 && 0 \le o'_2 \\
&\quad o_2 \le p_2 - 1 && \quad o'_2 \le p_2 - 1 \\
&o_2 \le o_1 && o'_2 \le o'_1 \\
&\quad o_1 \le o_2 + p_1 - 1 && \quad o'_1 \le o'_2 + p_1 - 1 \\
&\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = L \cdot \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} + \begin{pmatrix} o_1 \\ o_2 \end{pmatrix} && \begin{pmatrix} x'_1 \\ x'_2 \end{pmatrix} = L \cdot \begin{pmatrix} t'_1 \\ t'_2 \end{pmatrix} + \begin{pmatrix} o'_1 \\ o'_2 \end{pmatrix} \\
&& x'_1 = x_1 + 2 \\
&& x'_2 = x_2 + 1
\end{aligned}
\tag{5.4}
$$

If we wanted to generate code for the communication between the tiles, we would have to calculate two separate pieces of information: we would need to compute which tile a given tile sends its data to (variable ordering $t_1, t_2, t'_1, t'_2, \ldots$), and which tiles a given tile receives data from (variable ordering $t'_1, t'_2, t_1, t_2, \ldots$). We discuss here only the calculation for the sender code. An appropriate variable ordering for this problem is $t_1, t_2, t'_1, t'_2, x_1, x_2$ (projecting away the other dimensions). If we look at the inequalities of t'_1 and t'_2, we get an enumeration of the destination tiles in dependence of t_1 and t_2. And, for every (t'_1, t'_2), we have an enumeration of the source points (x_1, x_2) whose data must be sent to (t'_1, t'_2). (The question of which points inside the target tile need data from which source point is solved on the receiver side).

Projecting System (5.4) to the variables $t_1, t_2, t'_1, t'_2, x_1, x_2$ *without* calculating bounds for t_1 and t_2 (i.e., the elimination stopped after finding the bounds for t'_1 in dependence of t_1 and t_2) took 36 seconds using REDLOG for tree simplification, and 87 seconds when we used our CAD implementation to simplify the decision tree.

Computing also the bounds for t_1 and t_2 took considerably longer. We could not wait for an answer to the whole problem; we had to specialize System (5.4), for example with $p_2 = 4$. In this case, the full projection took 450 seconds using REDLOG for tree simplification, and 294 seconds with the CAD based simplifier. The sorting algorithm from Section 4.3.2 could not deliver a result at all in any case, it overflowed REDUCE's maximum heap size of 128 MB after two hours of computation.

5.3 Observations

During the experiments reported in this chapter and other applications of our procedures, we made some general observations concerning our implementation. We briefly summarize these observations:

- We consider the use of REDLOG's quantifier elimination features as a decision method for logical formulas to simplify decision trees very successful. Most of the problems were solved in less than one second (including the startup time for REDUCE).

- Our CAD implementation was sufficient to simplify the decision trees in the examples of this chapter. The performance was (as expected) most of the time not as good as REDLOG's, but it was sufficient to get results in reasonable time. The examples we chose are "friendly" in the sense that the polynomials appearing in the decision trees do not have a high total degree (usually less than 3) and at most 3 parameters. This is a reason for the good performance of our CAD implementation, since we observed severe performance degradation with inputs of polynomials with a total degree of 4 and higher.

- There is some potential for better performance of the CAD approach: the simplification of the decision trees, as shown in Section 4.2.2, adds formulas to the context during the recursion over the decision tree. This means that, when the simplifier descends a branch of the tree, the set P of polynomials in the context is extended to a new set $P' \supseteq P$ in each step. Our CAD implementation computes a fresh decomposition of P' without taking advantage of the decomposition already computed for P. In all three steps of the CAD procedure (projection, base case, extension) some work is repeated, since some of the polynomials and zeros are the same. We have not optimized our implementation to avoid these redundant computations, since this would corrupt its modular structure. This kind of optimization is not possible with REDUCE, since we have no access to the internal state of its quantifier elimination procedure.

- The use of quantifier elimination with answer seems to be problematic, since the algorithms relying on it tend to produce huge outputs (see the convex union example and the simple tiling example) or need more computation time than the approach using Fourier-Motzkin elimination (see the tiling examples) and sometimes run out of memory.

These points suggest some topics for further research:

- Currently, our implementation starts a new REDUCE process every time a quantifier elimination problem has to be solved. Although the start-up of REDUCE is very fast, a closer connection between REDUCE and our system could eliminate some of the invocation costs for every quantifier elimination call.

- An analysis of the decision problems occurring in practical problems during decision tree simplification could lead to a CAD implementation which is optimized for the situations which appear frequently when using our methods in real-world applications.

- The use of quantifier elimination to solve some problems directly requires further investigation on how the questions we ask the quantifier elimination with answer method can be reformulated such that the algorithms construct smaller results in less computation time.

Chapter 6

Conclusion

We have demonstrated that we can extend the existing polyhedron model with linear parameters to a more general model with non-linear parameters. The main difference to the classical polyhedron model is that, in the generalized model, one cannot determine the signs of every coefficient in a given inequality system statically. The consequence is that algorithms which do not require case distinctions in the classical case now produce results with case distinctions. A good example of this difference is the Fourier-Motzkin elimination method, which is described in Sections 2.1.2 and 4.2.1.

The challenge which arises in this enriched model is to simplify the decision trees representing the case distinctions. We chose to achieve this simplification by using quantifier elimination to check whether some condition implies another condition. We decided to use two different implementations of quantifier elimination: the REDLOG package of the computer algebra system REDUCE and our own implementation of a decision method based on cylindrical algebraic decomposition (CAD). Since our CAD implementation is not highly optimized, the performance is better when using REDLOG, in most cases, but the CAD implementation is good enough to solve some not too complex problems in reasonable time.

We discovered that we can use quantifier elimination in two different ways to obtain algorithms for the generalized polyhedron model. The first of the two approaches is to transform an existing algorithm for the classical polyhedron model into an algorithm which can handle non-linear parameters. We outlined this procedure in Section 4.2, and showed a generalized version of Fourier-Motzkin elimination as an example of this approach (Section 4.2.1). Algorithms generalized this way can produce large case distinctions (on the signs of parametric coefficients). Our implementation offers REDLOG or our CAD implementation to simplify these case distinctions top-down using the procedure shown in Section 4.2.2.

Working with REDLOG suggested a second approach to get generalized algorithms: expressing some problems directly as first-order logical formulas and using quantifier elimination to solve these problems. This enables us to solve some more problems. We outlined how to compute lexicographic minima and maxima (Section 4.3.1) and convex unions of polyhedra (Section 4.3.3). The convex union algorithm uses the method to calculate a sorted description of an inequality system which is equivalent to applying Fourier-Motzkin described in Section 4.3.2. Although we spent quite some time tuning the implementation of this method and tried some variants, it turned out that good performance is not easy to achieve. The main problem seems to be that REDLOG has been designed as a general quantifier elimination tool, so it cannot take advantage of special properties of the input. We frequently observed that quantifier elimination results got very big (compared to the input),

	#Vars	#Params	#Vars in projection	FM + REDLOG	FM + CAD	QE-Sort
2-dim. Tiling	6	3	4	5 sec.	9 sec.	10 sec.
Communication in 2-dim. Tiling	12	3	4	36 sec.	87 sec.	—
2-dim. Tiling & Comm. partners	12	2	12	450 sec.	294 sec.	—

Table 6.1: Summary of experiments conducted with our sorting algorithms

and this often leads to big decision trees carrying complex conditions with up to some thousands of atomic formulas. More investigation is required on which (of the many) switches controlling REDLOG's behavior should be activated or deactivated and how the problems can be formulated differently so REDLOG can find simple descriptions of the results more easily.

The sample applications in Chapter 5 show that our existing implementation can be used to solve some simple problems with non-linear parameters. Table 6.1 shows a summary of the running times of some of the experiments we conducted. It compares the performance of our generalized Fourier-Motzkin algorithm (denoted by "FM") together with REDLOG or CAD as decision method used by the tree simplifier and the sorting algorithm of Section 4.3.2 (denoted by "QE-Sort") which uses REDLOG's quantifier elimination with answer. The table lists the number of variables and parameters in the input system and the number of variables on which we project the input system to solve the respective problem. The table shows that the sorting algorithm based on quantifier elimination only works for small inputs and it is slower than Fourier-Motzkin, unfortunately. The Fourier-Motzkin implementation is cappable of sorting bigger systems which have nearly real-world application size. Our CAD implementation was faster than REDLOG in one of the shown experiments; this is mainly due to the fact that the respective experiment generates 3413 calls to the quantifier elimination procedure and the start-up costs for the REDUCE algebra system are relatively high compared to the complexity of the decision problems occurring in this example.

Beside the sorting algorithms, we also considered algorithms for convex and disjoint unions of polyhedra. The usability of the convex union algorithm is restricted by its dependence on the quantifier elimination based sorting algorithm. Therefore, it cannot be used for real-world problems currently. Our method to compute disjoint unions of polyhedra depends mainly on the computation of disjunctive normal forms of logical formulas and is much faster: the computation time for the example in Section 5.1.2 is less than one second.

We expect that further research in the area of non-linear parameters will bring improvements of the algorithms and the implementations, and we hope that finally problems like parametric tile sizes in automatic loop parallelization (which originally spurred our interest in non-linear parameters) can be handled for real-world inputs.

Bibliography

[ACM98] Dennis S. Arnon, George E. Collins, and Scott McCallum. Cylindrical Algebraic Decompositions I: The Basic Alogrithm. In Bob F. Caviness and Jeremy R. Johnson, editors, *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 136–151. Springer-Verlag, 1998.

[AI91] Corinne Ancourt and François Irigoin. Scanning Polyhedra with DO Loops. *Third ACM SIGPLAN Symposium on Priciples & Practice of Parallel Programming*, 26(7):39–50, July 1991.

[Ban93] Utpal Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, 1993.

[BW93] Thomas Becker and Volker Weispfenning. *Gröbner Bases: A Computational Approch to Commutative Algebra*. Springer-Verlag, New York, 1993.

[DS97a] Andreas Dolzmann and Thomas Sturm. Redlog: Computer algebra meets computer logic. *ACM SIGSAM Bulletin*, 31:2–9, June 1997.

[DS97b] Andreas Dolzmann and Thomas Sturm. Simplification of quantifier-free formulae over ordered fields. *Journal of Symbolic Computation*, 24:209–231, August 1997.

[FCB02] Paul Feautrier, Jean-François Collard, and Cédric Bastoul. Solving systems of affine (in)equalities. Technical report, PRiSM, Versailles University, 2002.

[GL97] Martin Griebl and Christian Lengauer. The loop parallelizer LooPo—Announcement. In David Sehr, Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing (LCPC'96)*, LNCS 1239, pages 603–604. Springer-Verlag, 1997. More details at `http://www.fmi.uni-passau.de/loopo`.

[Hea99] Anthony C. Hearn. REDUCE User's Manual, Version 3.7, July 1999. `http://www.zib.de/Optimization/Software/Reduce/moredocs/reduce.pdf`.

[Hon98] Hoon Hong. An Improvement of the Projection Operator in Cylindrical Algebraic Decomposition. In Bob F. Caviness and Jeremy R. Johnson, editors, *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 166–173. Springer-Verlag, 1998.

[HPF00] Paul Hudak, John Peterson, and Joseph Fasel. Gentle Introduction To Haskell, version 98, June 2000. `http://haskell.org/tutorial/`.

[Lam74] Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.

[Len93] Christian Lengauer. Loop Parallelization in the Polytope Model. In Eike Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer-Verlag, 1993.

[Loo83] Rüdiger Loos. Computing in Algebraic Extensions. In Bruno Buchberger, George E. Collins, and Rüdiger Loos, editors, *Computer Algebra, Symbolic and Algebraic Computation*, pages 173–187. Springer-Verlag, New York, second edition, 1983.

[LW93] Rüdiger Loos and Volker Weispfenning. Applying Linear Quantifier Elimination. *The Computer Journal*, 36(5):450–462, 1993. Special issue on computational quantifier elimination.

[PJ03] Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

[Red] `http://www.zib.de/Symbolik/reduce/`.

[Sch94] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1994.

[Tar51] Alfred Tarski. A Decision Method for Elementary Algebra and Geometry, University of Califonia Press, 2nd edition, revised, 1951.

[vzGL02] Joachim von zur Gathen and Thomas Lücking. Subresultants revisited. *Theoretical Computer Science*, 297(1–3):199–239, March 2002.

[Wei88] Volker Weispfenning. The Complexity of Linear Problems in Fields. *Journal of Symbolic Computation*, 5(1&2):3–27, February–April 1988.

[Wei94] Volker Weispfenning. Quantifier elimination for real algebra—the cubic case. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC 94)*, pages 258–263. ACM Press, July 1994.

[Wei97] Volker Weispfenning. Quantifier elimination for real algebra—the quadratic case and beyond. *Applicable Algebra in Engineering Communication and Computing*, 8(2):85–101, February 1997.

[Wie97] Martina Wieninger. Partitionierung von parallelen Schleifenprogrammen. Diplomarbeit, Universität Passau, 1997.

[Wil93] Doran K. Wilde. A library for doing polyhedral operations. Technical Report 785, IRISA, 1993.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Diplomarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet habe, sowie dass diese Diplomarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt wurde.

Passau, den 23. September 2003

(Armin Größlinger)