Masterarbeit im Studiengang Informatik

Using Single CAD Cells as Explanations in MCSAT-style SMT Solving

von
Heinrich-Malte Neuß
Matrikelnummer 307877
malte.neuss@rwth-aachen.de

vorgelegt der Fakultät für
Mathematik, Informatik und Naturwissenschaften der
Rheinisch-Westfälischen Technischen Hochschule
Aachen

im September 2018

Erst- und Zweitprüfer:
Prof. Dr. Erika Ábrahám
Prof. Dr. Viktor Levandovskyy

Betreuer:
M.Sc. Gereon Kremer

angefertigt am
Lehr- und Forschungsgebiet Informatik 2
Theory of Hybrid Systems
Universitätsprofessorin Erika Ábrahám

Contents

1 Introduction

Satisfiability Modulo Theories (SMT) comprises a family of low level problem modelling languages that vary in degrees of expressiveness and in theoretical solving efficiency. SMT solving is used to formally describe and solve problems in a variety of domains and can be summarized as either finding suitable values for variables to satisfy a combination of constraints—formally called a "formula"—or returning a proof that the given combination of constraints is unsatisfiable. This is the "Satisfiability"-part in SMT, while the "Modulo Theories"-part defines the shape and expressiveness of the constraints. In this thesis we focus on solving formulas from the theory of "Non-Linear Real Arithmetic" (NRA). Examples of such formulas are

$$x^2 - 1 < y \quad \wedge \quad y \geq 4, \tag{1}$$

and

$$x = 3 \quad \Longrightarrow \quad (x^2 y - y^2 + 2 < xy \quad \vee \quad \neg(y^2 - 3 \geq 0)). \tag{2}$$

In NRA the constraints have the shape of equalities and inequalities between polynomials—the "Non-Linear"-part in NRA—and the variables accept real numbers $\mathbb{R}$—the "Real"-part in NRA. These polynomial constraints are combined with the usual logical operators $\neg$ (logical NOT), $\wedge$ (logical AND), $\vee$ (logical OR) and $\Longrightarrow$ (logical IMPLIES). Solving such formulas is of great interest because they arise in important domains such as automated program analysis and verification [Con+05]. Having a NRA-formula, "solving" consists of either finding a satisfying assignment, which is a mapping for each of the formula's variables to a real number that makes the formula "true", or proving that there exists no satisfying assignment. For example, a satisfying assignment for Eq. 1 is $\{x \to 0,\ y \to 4\}$, because it satisfies each of its two constraints, while a formula like

$$x < 1 \quad \wedge \quad x \geq 2,$$

is unsatisfiable, because the second constraint implies $x \geq 1$, which is the exact opposite of the first constraint.

To search for a satisfying assignment, not knowing whether there exists one, we use a guided "Guess-And-Check" strategy called MCSAT, which roughly works as follows: We incrementally guess and assign a value to a yet-unassigned variable, and check whether this assignment comes into conflict with previous variable assignments in some constraint. If this indeed creates a conflict, we take back one or more variable assignments and guess other values instead. Once we have exhausted all possible assignments, we have proven that there is no satisfying assignment and the formula in question is unsatisfiable.

The main problem in guessing real values is the infinite number of them, which can get us stuck in guessing and taking back different but useless values indefinitely. For example, let's assume that we guessed $\{x \to 0\}$ for the formula in Eq. 1 and checked that there was no conflict. Afterwards we could guess and assign $\{y \to 0\}$. However, the combined assignment

$$\{x \to 0,\ y \to 0\}$$

would satisfy the first constraint but lead us to a conflict in the second. This second contraint would force us to take back our decision for y and guess again.

Unfortunately, we could continue to guess useless values for y such as 0.1, 0.11, 0.111 and so on. These values behave identically in not satisfying the second constraint and could keep us stuck. In this formula suitable values for y are obvious, but in a more complicated formula they are not.

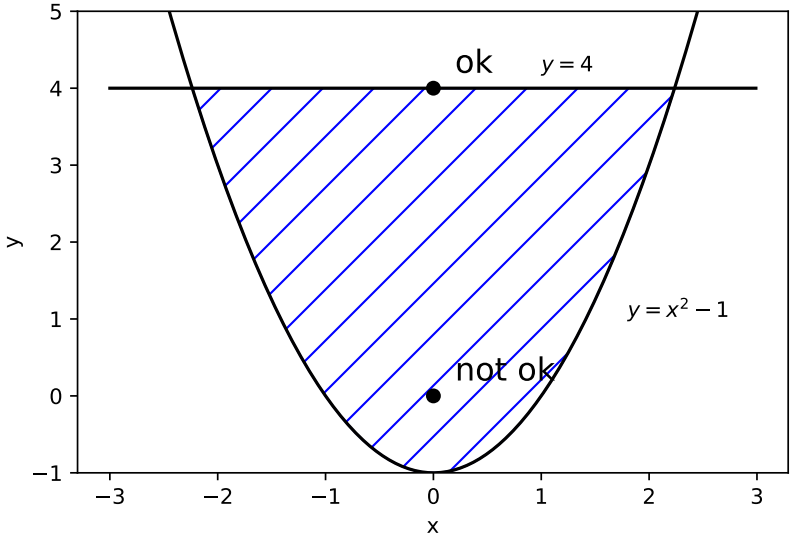


Figure 1.1: Graph of points (x, y) that satisfy the polynomial equalities $x^2 - 1 = y$ and $y = 4$. We see the geometrically-connected region—not including the solid lines—around point $(0, 0)$. This point and all points in the region behave similarly in not satisfying the formula in Eq. 1.

The main focus of this thesis is a technique to reduce the number of guesses to a finite amount by computing similar-behaving, geometrically-connected regions that exclude a great number of useless assignments at once. As depicted in Fig. 1.1, every assignment of variables x, y for the formula in Eq. 1 corresponds to a point on the x-y-plane, formally the two-dimensional real space $\mathbb{R}^2$. Given an assignment-point, we want to compute a connected, "sign-invariant" region—a concept, that is introduced in Sec. 2.1—which contains that point and as many other points as possible that behave similarly: the points all simultaneously satisfy a formula or they all simultaneously don't.

For example, if we compute the region around point $(0, 0)$ as in Fig. 1.1 and check that this point does not satisfy the formula in Eq. 1, we can ignore all other points in that region as well. So, sign-invariant regions guide us away from many useless assignments. As it turns out, for every formula there exists a decomposition of its variable-assignment space—one of which is called CAD—into at most finitely many sign-invariant regions and such a decomposition enables us to make at most finitely many guesses.

A performant SMT solving framework, that facilitates the interplay between guessing and checking real values for variables, is called "Model Constructing Satisfiability Calculus" (MCSAT); a "model" is the mathematical-logic word for a satisfying assignment we intend to find. It has been established by de

Moura and Jovanović [MJ13] and its efficiency greatly depends on the quality of our sign-invariant regions: To reduce the number of guesses and get the best possible performance, these regions should be as large and as efficient to compute as possible.

One of the best algorithms to compute invariant regions in multidimensional real space is called "Cylindrical Algebraic Decomposition" (CAD) [ACM84]. It decomposes the whole space into so-called "cylindric algebraic cells", which are approximations of the actual sign-invariant regions: A cell may be smaller than the real sign-invariant region. Furtheremore, a cell looks like a cylinder, in a way that we present in Sec. 2.1; this is the "Cylindric"-part in CAD. Finally, the cell's boundaries are represented by roots of polynomials, which is the semi-"Algebraic"-part in CAD.

CAD is one of the few effective methods to compute sign-invariant regions. However, computing a full decomposition of the whole multidimensional real space is computationally expensive and is often unacceptably slow even though we always end up with finitely many cells. One reason is that the finite number of computed cells can be an unacceptably large amount. Also, CAD scales computationally bad in the number of processed polynomials and scales even worse in the number of variables that appear in those polynomials—doubly-exponential to be precise [BD07]. Fortunately, we can ease many of these pain points with MCSAT in the context of SMT

First, we don't need a full decomposition at once. Instead we construct single CAD-cells iteratively on demand. Once we find the first satisfying assignment-point in an invariant region, we ignore the invariant regions in the remaining, unexplored real space, which saves us computation time.

Second, since we have disjunctions and negations ($\vee$- and $\neg$-combinations) in a formula, not all constraints will be "active" at all times. In Eq. 2 for example, once we assign a value other than 3 to x, the left-hand side of the implication $\implies$ is false and thus none of the constraints of the right-hand side will matter. Therefore, in the context of SMT we can use CAD with a subset of polynomials—those from the "active" constraints— and this subset may mention only a subset of all variables—not every polynomial mentions every variable. This also saves us computation time.

In the context of SMT a logical formula that is a representation of a conflicting assignment is called a "conflict-explanation", or simply "explanation"; it "explains" a previously unkown, implicit conflict by making it explicit. In MCSAT this formula includes a representation of a sign-invariant region around this conflicting assignment, which can be represented as a point in the real space. In our implementation we use a CAD-cell as an approximation of such a region—the cell "explains" and generalizes the conflicting point as being part of more general conflicting region.

The MCSAT framework by Jovanović and de Moura has already been successfully implemented and combined with CAD in the context of non-linear formulas [JM13]. They were the first to create a more efficient CAD-variant to construct single cells instead of a full decomposition and to use it in SMT explanations. We want to improve their approach by using a novel, even more efficent, single-cell constructing CAD-variant, which theoretically produces larger cells in less time.

The contributions of this thesis can be summarized as follows:

- We summarize the background information that is necessary to understand the popular CAD and MCSAT publications (see Sec. 2 and Sec. 3). We target the knowledge level of a typical computer science student.
- We implement a novel single-CAD-cell constructing algorithm called "OneCell" by Brown and Košta [BK15] in C++ (see Sect. 2.5). This CAD-variant is in theory more efficient than Jovanović and de Moura's CAD-variant.
- We embed our "OneCell" implementation into the SMT-RAT[1] framework as an "explanation" backend for solving formulas with NRA constraints (see Sect. 3.4). SMT-RAT is a competitive SMT solving framework that belongs to the RWTH Aachen University and includes an MCSAT implementation based on [MJ13].
- We evaluate the performance of the embedding of our "OneCell" implementation into SMT-RAT against several other SMT solvers, especially against an existing single-cell CAD algorithm already embedded into SMT-RAT (see Sect. 4). For the evaluation we use the popular QF_NRA benchmark set that belongs to the SMT-LIB initiative[2].

[1] `https://smtrat.github.io/`
[2] `http://smtlib.org/`

2 Cylindric Algebraic Decomposition

Cylindrical Algebraic Decomposition (CAD) is the focus of this thesis and the main tool to deal with the polynomials inside the constraints of a formula from the theory of Non-linear Real Arithmetic such as

$$\underbrace{x^2 < -y^2 + 2}_{\text{constraint}} \quad \wedge \quad \underbrace{xy > 1}_{\text{constraint}} ,$$

which can always be rewritten as constraints against zero:

$$\underbrace{\underbrace{x^2 + y^2 - 2}_{\text{poly}} \underbrace{< 0}_{*}}_{\text{constraint}} \quad \wedge \quad \underbrace{\underbrace{xy - 1}_{\text{poly}} \underbrace{> 0}_{*}}_{\text{constraint}} . \tag{3}$$

Our goal is to find values over the reals $\mathbb{R}$ for the formula's variables such that all constraints are satisfied. The assignments for the variables can be seen as points in the multi-dimensional real space. In the notation from Eq. 3 we can see that every constraint basically contains one polynomial. Also, whether a (multi-dimensional) point satisfies a constraint solely depends on the sign—negative below zero, zero itself or positive above zero—of the constraint's polynomial at that point. This is exactly where CAD comes into play.

Most importantly, a CAD algorithm works solely on polynomials and uses their signs; It has no notion of constraints or "sign-conditions", which are marked with * in Eq. 3. However, this is no problem, because we already made the link from a polynomial's sign to a constraint's satisfication above. So, CAD takes a set of polynomials and decomposes the space of their variables into connected, regions called "cells", where the polynomials are "sign-invariant". This is the "Decomposition"-part in CAD. These cells happen to look a lot like cylinders—the "Cylindrical"-part—, and are represented by polynomials—the "Algebraic"-part in CAD. If we take a point from such a region and plug it into the polynomials, we can compute signs, one for each polynomial. Sign-invariance for this region means that if we take another point from the same region and plug it into the polynomials, we will compute the same sign for the same polynomial as before.

So in order to use CAD on a formula, we need to extract the polynomials from its constraints. The polynomials in Eq. 3 are:

$$\{x^2 + y^2 - 2, \quad xy - 1\}. \tag{4}$$

CAD then constructs sign-invariant regions with the following property: If we take one point from a region, plug it into one constraint and see that it does not satisfy it—the constraint's polynomial gets the wrong sign—, then we can avoid all other points in that region, because they also won't satisfy the constraint. And as a result of the region being invariant in all the formula's polynomials, this property holds for the formulas as well: If we check one point on a formula, we don't need to check any other point from that region. This makes CAD tremendously powerful!

2.1 Intuitive Geometric CAD

Since CAD is formally quite complicated, we begin with an intuitive introduction before we apply more mathematical rigour. Fortunately, sign-invariant cells have a nice geometric interpretation.

2.1.1 Sign-invariant regions in visualizations

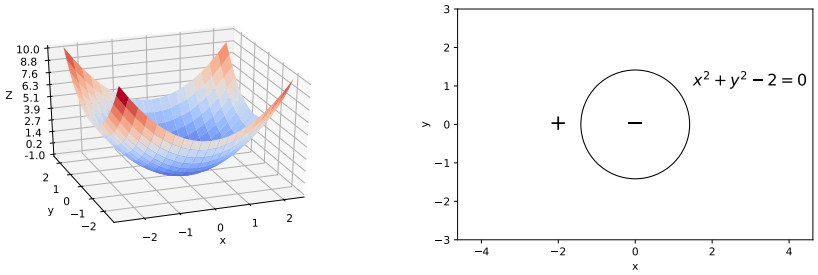


Figure 2.1: Polynomial $x^2 + y^2 - 2 = Z$ (left) and its root plot $x^2 + y^2 - 2 \stackrel{!}{=} 0$ with its 3 sign-invariant regions (right).

First of all, we have to get a feeling for how we show a polynomial's sign-invariant regions visually. Take for example the polynomial $x^2 + y^2 - 1$ from Eq. 4. in Fig. 2.1. On the left we see that it's a scalar function from (x, y) values to a Z value. Therefore we need a 3D image to show the Z values. However, considering only being interested in the signs of those Z values, a 2D image, as on the right, for the (x, y) values is sufficient. This 2D image is the cut through the 3D plot at $Z = 0$: We see the polynomial's roots—the (x, y) values where the polynomial becomes zero—in solid lines. This is why we call this a "root plot". Also, the regions between and outside of the roots are marked with $+$ and $-$ to represent the sign of the Z values. We can see that these regions are connected, sign-invariant—within each region we have the same Z sign—and always separated by roots, which form connected, sign-invariant regions themselves: We get three regions in total.

In general a polynomial with n variables requires an n-dimensional root plot, which is why we will present polynomials with at most 3 variables. You can find another root plot example for the other polynomial in Eq. 4 in Fig. 2.2.

2.1.2 Cylindric regions

Now we can get an intuition for what regions a CAD algorithm computes and how these are cylindrical. In Fig. 2.3 we see a CAD decomposition for polynomial $x^2 + y^2 - 2$ from Eq. 4 into 13 regions, which is much more than optimally necessary, because there are only three actual invariant regions as seen in Fig. 2.1. The reason is that CAD can't do better because it can only construct "cylindrical" regions. As such, CAD produces only an approximation. Nevertheless, we may agree that regions 2, 3, and 9 look somewhat cylindrical, but it is less obvious why regions 1 and 6 are considered cylindrical.

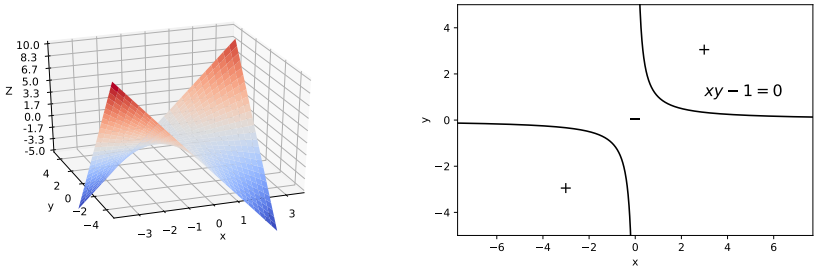


Figure 2.2: Graph of polynomial $xy - 1 = Z$ (left) and its root plot $xy - 1 \stackrel{!}{=} 0$ with its 5 sign-invariant regions (right).

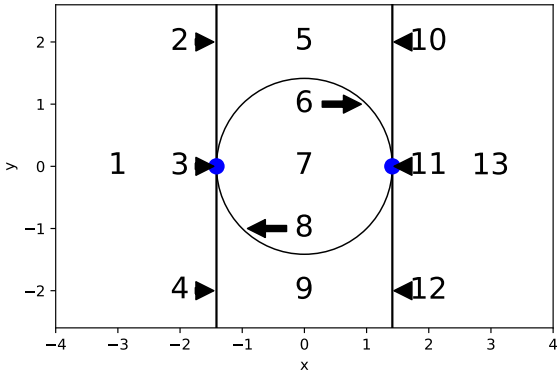


Figure 2.3: Root plot of polynomial $x^2 + y^2 - 2 \stackrel{!}{=} 0$ (solid circle) and a CAD decomposition with numbered cells.

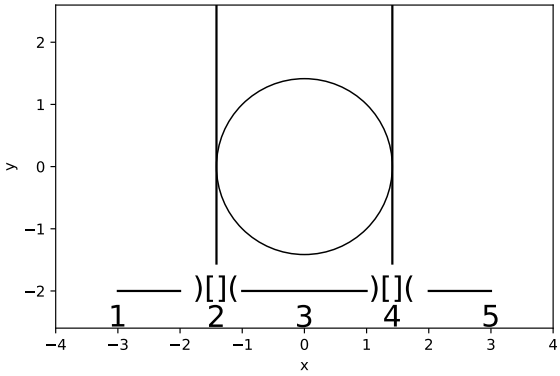


Figure 2.4: Root plot of polynomial x^2+y^2-1. We see 5 1-dimensional cylinders along the x-axis and their extensions along the y-axis to 2-dimensional cylinders.

Intuitively, these regions are cylindrical because they emerge from cylinders: We first create 1-dimensional cylinders on the x-axis. These are either open intervals or closed, single-point intervals as in Fig. 2.4. Then from each 1-dimensional cylinder we create a 2-dimensional one by extending it along the y-axis, which is called "extension" or "lifting". Then each 2-dimensional cylinder is separated into fragments by the polynomial's roots that cross it. These fragments are the regions—called CAD-cells—we see in Fig. 2.3. In short, we call these regions cylindrical, because they are fragments of cylinders. Another more formal characterization of cylindric is that the projection of all the cells within a cylinder—by dropping the y–coordinate of the points within them—is the same 1-dimensional region along the x-axis.

Each cylindrical region has two "components": The 1-dimenional cylinder that constraints it on the x-axis, and the root-segments of the 2-dimensional polynomial that constraints it on the y-axis (see regions 9 in Fig. 2.3 and Fig. 2.4). A CAD algorithm constructs a 1-dimensional cylinder on the x-axis in the following way: whatever point x on this cylinder we select, we will cross the same number of the polynomials' roots in the same order as we move this point along the y axis. This property is called "delineability" and ensures that we will be able to formally represent a cell as a sequence of separate components, which is necessary for a compact and efficient representation. A cell's first two components along the x and y-axis will have the form

$$(\text{constant} < \quad x \quad < \text{constant}) \text{ and } (f(x) < \quad y \quad < g(x)),$$

To find the 1-dimensional cylinders in the example above, a CAD-algorithm uses a clever recursive scheme: From 2-dimensional polynomials—with x and y variables—it creates a set of 1-dimensional polynomials—with only the x variable. It creates these polynomials in such a way that their roots—real values on the x-axis—will become the bounds of these 1-dimensional interval-cylinders as in Fig. 2.4. This poly-set creation is called a "projection". As we will see in the following, creating those 1-dimensional intervals is actually a decomposition of the 1-dimensional real space into cylindrical, sign-invariant regions.

2.1.3 Decomposition of the 1-dimensional real space

Let's assume that we have a set of 1-dimensional polynomials like

$$\{x^2 - 2\}$$

with a single polynomial for simplicity. In Fig. 2.5 the right graph shows that the root plot only needs to be 1D and that the polynomial's roots at $\pm\sqrt{2}$ separate the real line along the x-axis into connected, sign-invariant regions. So by computing all the roots, which is known as "root isolation", we can construct these regions as follows: Each root makes up a single-point, closed interval region, a "section", and each interval between two consecutive roots makes up an open regions, a "sector". We consider them cylindrical by definition. However, more intuitively they emerge as fragments from the 1-dimensional cylinder along the x-axis—from $-\infty$ to $+\infty$. Finally, the set of all those open and single-point intervals, our 1-dimensional CAD-cells, is a CAD of $\mathbb{R}^1$.

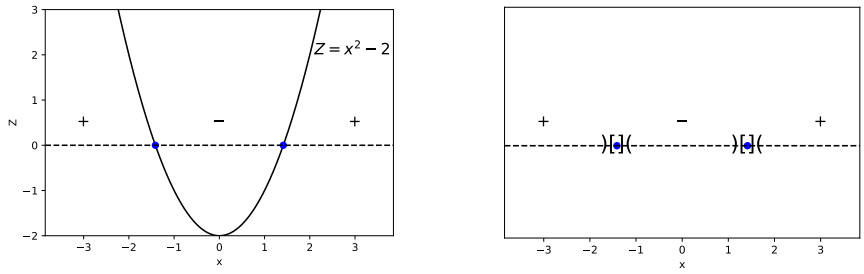


Figure 2.5: Graph of polynomial $x^2 - 2 = Z$ (left) and its root plot $x^2 - 2 \stackrel{!}{=} 0$ with its 5 sign-invariant regions (right).

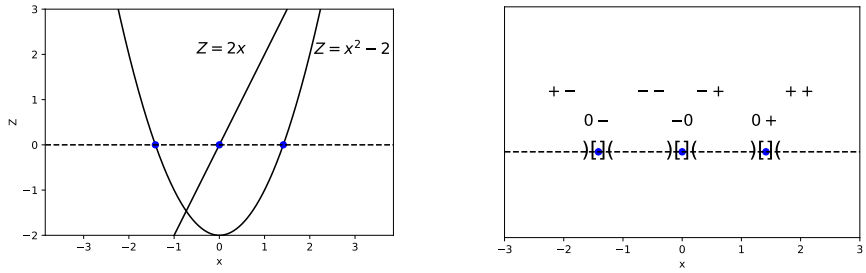


Figure 2.6: Graph of polynomials $x^2 - 2 = Z$ and $2x = Z$ (left) and their root plots $x^2 - 2 \stackrel{!}{=} 0$ and $2x \stackrel{!}{=} 0$, and 7 sign-invariant regions (right); The first sign in each region belongs to the first polynomial, the second sign to the second polynomial.

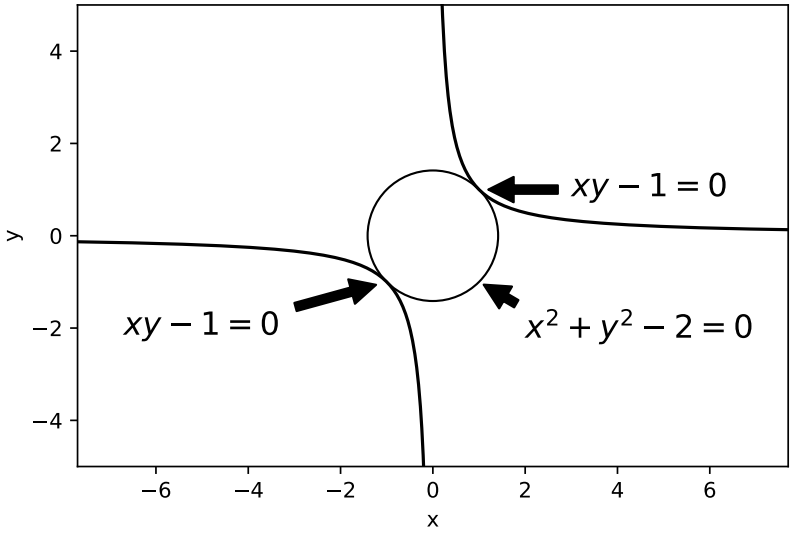


Figure 2.7: Root plot of polynomials $x^2 + y^2 - 2 \stackrel{!}{=} 0$ and $xy - 1 \stackrel{!}{=} 0$.

In the right graph of Fig. 2.6 we can see another CAD when there is a second, 1-dimensional polynomial: We see how the regions are sign-invariant for both polynomials simultaneously and that the sign-pairs always and only change at one of the polynomial's roots. To construct these regions we isolate all the roots of each polynomial, ignore that the roots came from different polynomials, and use them as before.

2.1.4 Decomposition of the 2-dimensional real space

Knowing how to construct a CAD for $\mathbb{R}^1$ allows us to construct a CAD of $\mathbb{R}^2$. In Fig. 2.7 we see a root plot of the set

$$\{x^2 + y^2 - 2, \quad xy - 1\}$$

of two 2-dimensional polynomials, which we already introduced in Sec. 2.1.1. Recall that to construct a CAD, we first must compute 1-dimensional regions along the x-axis such that for each point within it, as we move along the y-axis, the number and order of the polynomials' roots we cross doesn't change—compared to any other point of the same region. As it turns out, the number and order of roots change at the x-coordinates of the following special points: double roots along the y-axis, intersections of two polynomials and singularities, which we see in Fig. 2.8. A CAD algorithm will therefore transform the set of 2-dimensional polynomials into a set of 1-dimensional ones, whose roots will precisely be the x-coordinates of those special points.By isolating the roots of these 1-dimensional polynomials, we get 1-dimensional sign-invariant, cylindric regions—called cells from now on. We extend these cells into 2-dimensional cylinders along the y-axis into $\pm\infty$, which we then separate by the polynomials' roots that cross these 2-dimensional cylinders.

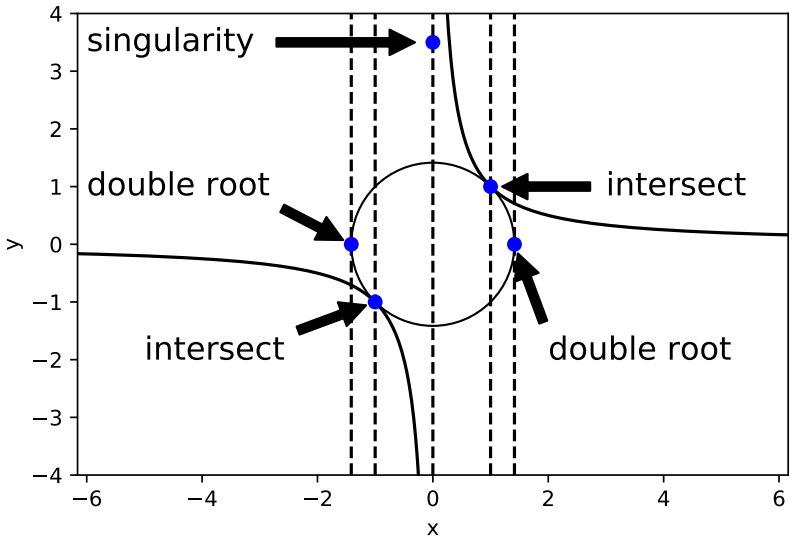


Figure 2.8: Root plot and special points—double roots, singularities and intersections—of polynomials $x^2 + y^2 - 2 \stackrel{!}{=} 0$ and $xy - 1 \stackrel{!}{=} 0$.

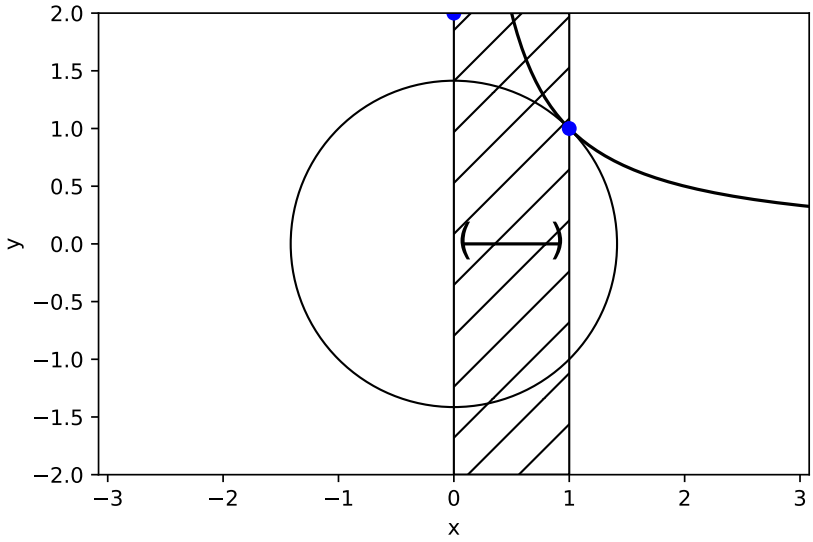


Figure 2.9: Extending the single, 1-dimensional cell $(0, 1)$ into a cylinder along the y-axis.

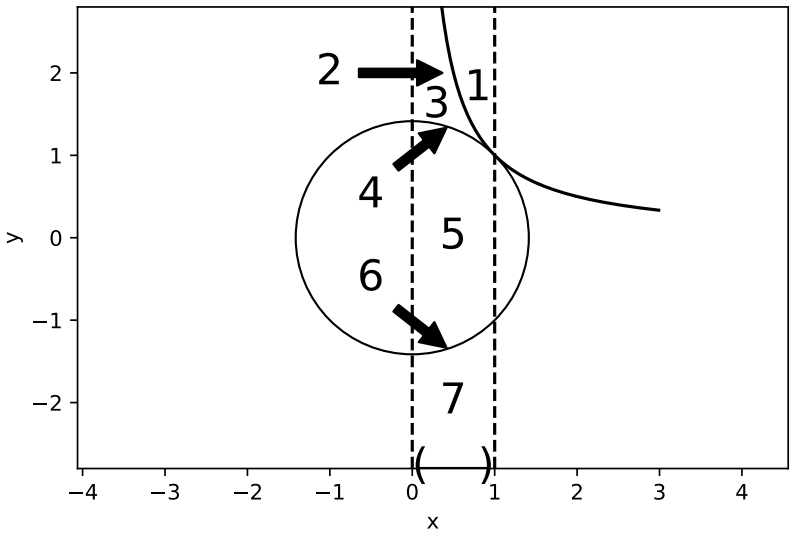


Figure 2.10: Cylinder over 1-dimensional cell $(0, 1)$ separated by polynomials' roots into 7 2-dimensional cylinder fragments we call CAD-cells.

In Fig. 2.9 we see an example of such an extension into a cylinder, which we then separate into cylinder fragments as in Fig. 2.10. The "open" fragments between two root-segments are called sectors and "closed" fragments on the root-segments are called sections. These fragments are our 2-dimensional CAD-cells, and the set of all these cells—emerging from all 2-dimensional cylinders—is a CAD of $\mathbb{R}^2$.

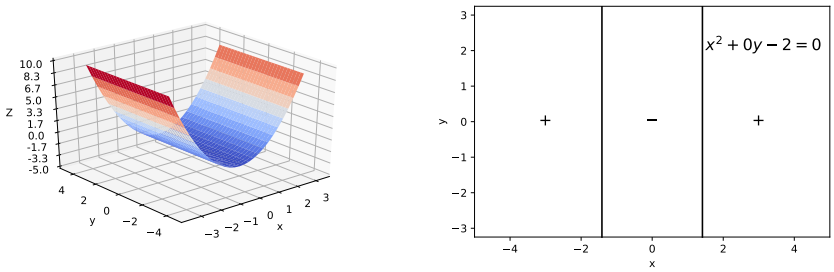


Figure 2.11: Graph of polynomial $x^2 + 0 \cdot y - 2 = Z$ (left) and its root plot $x^2 + 0 \cdot y - 2 \overset{!}{=} 0$ (right).

If we want to visualize polynomials with different numbers of variables together in one root plot, it is useful to know that we can reinterpret a 1-variable polynomial such as $x^2 - 2$ from Sec. 2.1.3 as the 2-variable polynomial

$$x^2 + 0 \cdot y - 2 < 0 \tag{5}$$

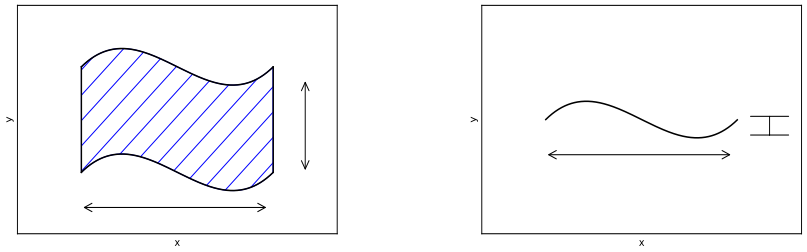
by adding the term $0 \cdot y$. In Fig. 2.11 we can see how to visualize it as a regular 2-variable polynomial.

2.1.5 Section and sectors

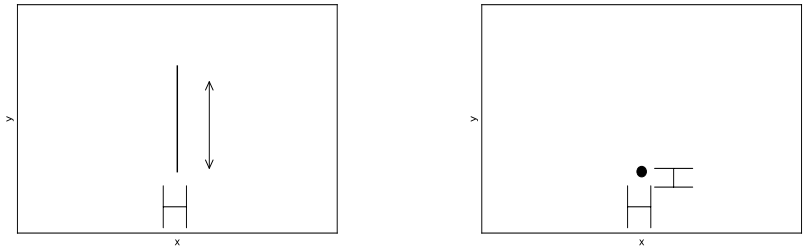
At this point it is useful to come back to the terms "section" and "sector". Just as a 2-dimensional vector like

$$\begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

has 2 components, one for each axis, a 2-dimensional cell has 2 components, either a section or a sector for each axis—an n-dimensional cell will therefore have n components. So far we used the word "dimension" for the real space $\mathbb{R}^2$ a cell is embedded in—we call it the "universe". However, since a cell is a space itself, it can have a dimension of its own—possibly lower than 2 as in Fig. 2.12. For a vector we can count the number of components to compute its dimension; for a cell we count the number of sectors, because only sectors are "open" along an axis and therefore contribute to the dimension. As such, the cell in Fig. 2.12a has dimension 2, because it is open (and therefore has a sector) along the x and and y-axis. In contrast, the cell in Fig. 2.12b is open along the x-axis but closed along the y-axis. Reversly, the cell in Fig. 2.12c has a closed section along the x-axis, but an open sector along the y-axis. Finally, the single-point cell in Fig. 2.12d has two section components: its dimension is 0.



(a) 2-dim cell with x-sector and y-sector (b) 1-dim cell with x-sector and y-section



(c) 1-dim cell with x-section and y-sector (d) 0-dim cell with x-section and y-section

Figure 2.12: Different dimensional cells in a 2-dimensional $\mathbb{R}^2$ universe.

2.2 Formal Analytic CAD

Cylindric Algebraic Decomposition (CAD) decomposes a real-valued multidimensional space into regions that are sign-invariant to a given set of multivariable polynomials. If those polynomials mention variables $x_1, \ldots, x_n$, CAD decomposes the real space $\mathbb{R}^n$ by working in 2 phases, which roughly act as follows:

1. **Projection**:
   Stepwise eliminate variables from the polynomials in the backward order $x_n, x_{n-1}, \ldots, x_2$ until we only have polynomials that only mention variable x_1. During each step $i = n, \ldots, 2$ new polynomials with at most variables $x_1, \ldots, x_i$ are created, which are processed in the next elimination-step as well.

2. **Lifting**:
   From the set of polynomials with only variable x_1 isolate the roots of each polynomial to compute 1-dimensional cells. Next, stepwise extend lower-dimensional cells into the next-higher dimensional cells until we have n-dimensional cells. During each step $i = 2, \ldots n$ lift the cells of dimension $i-1$ into i-dimensional cells by using all polynomials from the projection-phase that precisely mention the variables $x_1, \ldots, x_i$.

In order to describe in more detail how the "projection" and "lifting" phases work, we first have to introduce some notation.

2.2.1 Basic vocabulary about polynomials

Formally we look at polynomials with integer coefficients and real-valued variables, known as the integral polynomial ring $\mathbb{Z}[x_1, \ldots, x_n]$.

Definition 2.1 (Range Notation)**.** For a shorter notation we define ranges

$$\begin{aligned} \boldsymbol{a}_1^n &\coloneqq a_1, \ldots, a_n, \\ \boldsymbol{\alpha}_1^n &\coloneqq \alpha_1, \ldots, \alpha_n, \\ \boldsymbol{x}_1^n &\coloneqq x_1, \ldots, x_n \text{ and} \\ \mathbb{Z}[\boldsymbol{x}_1^n] &\coloneqq \mathbb{Z}[x_1, \ldots, x_n] \end{aligned}$$

Definition 2.2 (Monomial)**.** A monomial over variables $\boldsymbol{x}_1^n$ is any term of the form

$$x_1^{c_1} \cdot x_2^{c_2} \cdot \ldots \cdot x_n^{c_n},$$

where each $c_i \in \mathbb{N}_0$ is a non-negative integer.

In this thesis all polynomials are represented and constructed with monomials. For example, the following are all monomials over variables $[x, y, z]$:

$$1 \coloneqq x^0y^0z^0 \qquad x \coloneqq x^1y^0z^0 \qquad y \qquad z \qquad xy \qquad xz \qquad yz \qquad xyz \qquad xy^2z^3.$$

As we can see, a monomial is the product of variables with a non-negative exponent, where we leave out variables with the exponent 0 whenever we do not need to emphasize that it is an important part of the monomial.

Definition 2.3 (Polynomial)**.** A polynomial of $\mathbb{Z}[\boldsymbol{x}_1^n]$, which we also refer to as an $\boldsymbol{x}_1^n$-polynomial, has the form

$$a_1 \cdot m_1 + a_2 \cdot m_2 + \ldots + a_k \cdot m_k,$$

where each coefficient $a_i \in \mathbb{Z}$ is an integer and each m_i is a distinct monomial over variables $\boldsymbol{x}_1^n$.

Polynomials are sums of monomials that use the same set of variables and where each monomial may be further multiplied with an integer coefficient. For example, the following are polynomials of $\mathbb{Z}[x, y, z]$:

$$1x + 2y + 3z \qquad 4xy + 5yz + 6xyz \qquad 7xy^2z^3 + 8 \qquad 9 \qquad 0$$

Definition 2.4 (Type of Polynomial)**.** We categorize polynomials of $\mathbb{Z}[\boldsymbol{x}_1^n]$ into the following types:

- Zero-Polynomial: 0
- Constant-Polynomial: $0, \ldots, 9$ without any variable.
- Non-zero-Polynomial: Any polynomial not equal to zero.
- Non-constant-Polynomial: E.g. $x+1$, y^2-2, with some variable, i.e., every polynomial that is not a single constant.

As we can see, $\mathbb{Z}[x, y, z]$ includes polynomials with fewer than those three variables, even constant polynomials like 9 and the zero-polynomial that do not mention any variable. This allows us to reinterpret a $[x, y]$-polynomial such as xy as a $[x, y, z]$-polynomial and vice versa.

In the sections to come we use names for polynomials like p and q, which we use as macros to be replaced by their definition. In order to reason more clearly about these macros we define 4 kinds of equality in the style of mathematical logic.

Definition 2.5 (Equality)**.** We define four types of equality:

- $:=$ (Macro) Definition
- $=$ (Semantics) "Normal" Equality
- $\equiv$ (Syntax) Identical Equality
- $\overset{!}{=}$ (Shall-be) Imperative Equality

The symbol ":=" introduces new macros for a polynomials such as

$$p := x^2 - 4 \qquad\qquad q := x - x,$$

which are to be substituted immediately by their right-hand side. Thus, whenever we write an equation such as

$$p = q$$

we mean the macro-expanded equation

$$x^2 - 4 = x - x$$

Furthermore, q is not syntactically equal to the zero-polynomial ($q \not\equiv 0$ but semantically equal $q = 0$, because it represent the same function as the zero-polynomial, i.e., it maps all its inputs to zero. Both p and q are neither syntactically equal ($p \not\equiv q$) nor semantically equal ($p \neq q$). Instead p is semantically equal to e.g. $(x-2)\cdot(x+2)$. Also, whenever we write

$$p \overset{!}{=} 0$$

we mean that we *want* p to be equal to 0, although it's not by itself. In genereal, this is a notation for the set of all of p's roots. Notably, the subtle difference between $q \overset{!}{=} 0$ and $q = 0$ is crucial for subsequent polynomial operations, because in the first case we ask for its roots and in the second we state that q is (semantically) the zero-polynomial.

In all polynomial operations that are neceassary for a CAD algorithm we will need to categorize polynomials based on the variables that actually appear. For example, a polynomial of $\mathbb{Z}[x,y,z]\backslash\mathbb{Z}[x,y]$ cannot be interpreted as a polynomial of $\mathbb{Z}[x,y]$ alone, because it indeed mentions the variable z. To distinguish such polynomial sets, we define the "level" of a polynomial.

Definition 2.6 (Level Of Polynomial)**.** Given a variable order $x_1 \prec x_2 \prec \ldots \prec x_n$, the level of a polynomial $p \in \mathbb{Z}[\boldsymbol{x}_1^n]$ is the highest index i—in that order—of a variable x_i that appears with non-zero exponent and with a non-zero coefficient. If p mentions no variable—it's a constant-polynomial—, then its level is 0.

Unless stated otherwise, we implicitly assume the natural, upward variable order of $x_1, \ldots, x_n$. For example, the following $[x,y,z]$-polynomials have the following levels:

Polynomial	$\mathbf{2}$	$0\cdot x + \mathbf{2}$	$\boldsymbol{x}+2$	$\boldsymbol{y}^2$	$\boldsymbol{z}^3 + x$	$x\boldsymbol{y}+2$	$y\boldsymbol{z}+xy$
Level	0	0	1	2	3	2	3

In our implicit $x \prec y \prec z$ variable order x gets index 1, y the index 2 and z gets index 3—the determining variable is marked in boldface.

This notion of a polynomial's level allows us to specify the phases of a CAD algorithm with more detail: Given a set of polynomials of $\mathbb{Z}[\boldsymbol{x}_1^n]$ and a variable

$$
\begin{array}{lcr}
P_n & & \text{level-}n \text{ cells} \\
\downarrow \text{ project} & & \uparrow \text{ lift} \\
P_{n-1} & & \text{level-}n-1 \text{ cells} \\
\downarrow \text{ project} & & \uparrow \text{ lift} \\
\dots & & \dots \\
\downarrow \text{ project} & & \uparrow \text{ lift} \\
P_2 & & \text{level-2 cells} \\
\downarrow \text{ project} & & \uparrow \text{ lift} \\
P_1 & \overset{\text{base case}}{\longrightarrow} & \text{level-1 cells} \\
\\
P_0 \text{ discard} & &
\end{array}
$$

Figure 2.13: High-level CAD algorithm.

order $x_1 \prec x_2 \prec \ldots \prec x_n$, do the following three steps:

1. **Projection**:
   Categorize the polynomials according to their level into polynomial buckets $P_0, P_1, P_2, \ldots, P_{n-1}, P_n$, where bucket P_i contains only polynomials of level i. For $i = n, n-1, \ldots, 2$ in that backward order iteratively compute the "projection" of P_i, which creates polynomials of smaller levels $1, \ldots, i$—we explain "projection" in the next section. Add these polynomials according to their level into the buckets for the next iteration $i-1$. Finally, delete the constant-polynomials in P_0.

2. **Base case**:
   Using polynomials in P_1 isolate their roots to construct level-1 cells—along the x_1 axis.

3. **Lifting**:
   For $i = 2, \ldots, n-1, n$ in that upward order iteratively extend each level-$i-1$ cells into level-i cells by using polynomials in P_i.

This high-level overview of a CAD algorithm is visualized in Fig. 2.13. The CAD-algorithm we choose for this thesis is by Brown [Bro01] and in the following sections we explain how to project and how to lift according to this algorithm.

2.3 Projection Phase for a full CAD

A projection in the CAD algorithm has one purpose: It eliminates a variable from a set of polynomials by creating another set —using some polynomial operations to come—of polynomials that don't mention that variable anymore. Thus, it creates a subproblem that is easier to solve, that is, a polynomial-set that is easier to construct CAD-cells for. While doing so, a projection ensures that the subproblem's CAD-cells can be extended into cells for the main problem.

Brown's CAD algorithm [Bro01] defines that, for the projection of a polynomial-set, we need to compute the *leading coefficient* and the *discriminant* of each polynomial, as well as the *resultant* between any two distinct polynomials.

We first describe how these operations are computed and then show how they are used and why they are needed.

2.3.1 Operations on integral, real polynomials

Computing the leading coefficient and a discriminant are operations on a single polynomial, while computing a resultant is an operation on a pair of polynomials. All three operations are technically defined for single-variable polynomials only. We therefore first show how to compute these for polynomials in $\mathbb{Z}[x]$ and afterwards show how and why these also work for multi-variable polynomials in $\mathbb{Z}[\boldsymbol{x}_1^n]$.

The first and simplest operation is computing the leading coefficient. For example, a polynomial of $\mathbb{Z}[x]$ like

$$1x^2 + 2x + 3$$

has the coefficients $1, 2, 3$ and the leading coefficient 1. Notice how in our notation for the class of polynomials the second part $[x]$ determines the variables to appear in the monomials and the first part $\mathbb{Z}$ determines the type of coefficients.

Definition 2.7 (Coefficients of a Single-Variable Polynomial)**.** Given a polynomial

$$p \coloneqq a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \ldots + a_1 \cdot x^1 + a_0$$

of $R[x]$, where R is any algebraic structure, $a_i \in R$ and $a_n \neq 0$, then $\text{coeffs}_x(p) = \{a_n, a_{n-1}, \ldots, a_1, a_0\}$ are the coefficients and $\text{leadCoeff}_x(p) = a_n$ is called the leading coefficient. Furthermore, $\text{coeffs}_x^{\neq 0}(p)$ contains only those coefficients that are not identically 0.

Definition 2.8 (Degree of a Single-Variable Polynomial)**.** Given a polynomial

$$p \coloneqq a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \ldots + a_1 \cdot x^1 + a_0$$

of $R[x]$, where R is any algebraic structure, $a_i \in R$ and $a_n \neq 0$, then

$$\deg(p) \coloneqq n$$

is called the "degree of p".

In Def.2.7 we define the word "coefficent" for a single-variable polynomial. However, when dealing with a multi-variable polynomial, there are two possible interpretations: On one hand, in Def. 2.3 from a previous section we used the word "coefficient" to refer to the integer in front of a monomial. On the other hand, we can reinterpret a multi-variable polynomial of $\mathbb{Z}[\boldsymbol{x}_1^n]$ with integer coefficients as a single-variable polynomials $\mathbb{Z}[\boldsymbol{x}_1^{n-1}][x_n]$ with respect to variable x_n—it has polynomials of $\mathbb{Z}[\boldsymbol{x}_1^{n-1}]$ as coefficients. This is the interpretation we need when we compute the leading coefficient.

For example, a multi-variable polynomial of $\mathbb{Z}[x, y, z]$ such as

$$q \coloneqq yz + \boldsymbol{x}^2yz + \boldsymbol{x}^2y^2 + \boldsymbol{x}z + 3,$$

which has coefficients $1, 1, 1, 1, 3$, can be rearranged and reinterpreted—our desired variable x is marked in boldface—as the $\mathbb{Z}[y, z][x]$ polynomial

$$q = (yz + y^2) \cdot \boldsymbol{x}^2 \quad + \quad (z) \cdot \boldsymbol{x} \quad + \quad (yz + 3). \tag{6}$$

In this reinterpretation the polynomials $yz+y^2$, z and $yz+3$ are the coefficients and $\text{leadCoeff}_x(q) \coloneqq yz + y^2$ is the leading coefficient with respect to variable x. To avoid any confusion, we use a subscript to denote the variable x.

The discriminant is our second operation on a single polynomial. It's an indicator whether a quadratic polynomial equation of the form

$$ax^2 + bx + c \stackrel{!}{=} 0$$

has two roots collapsing into one. Normally the closed-form solution of the roots is given as

$$r_{1,2} \coloneqq x_{1,2} \coloneqq \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Here the two roots collapse into one whenever the discriminant—the term $b^2 - 4ac$ in this case—becomes zero. However, this is a special case for polynomials of degree 2. In general, when we have single-variable polynomial of a higher degree

$$p \coloneqq a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \ldots + a_1 \cdot x^1 + a_0 \in \mathbb{Z}[x], \quad a_n \neq 0$$

with its roots $r_1, \ldots, r_k$, then the discriminant is defined as the product

$$\text{discr}(p) \coloneqq a_n^{2n-2} \cdot \prod_{i<j} \underbrace{(r_i - r_j)^2}_{*},$$

where the whole product becomes zero if two roots overlap—marked with *.

When we have multi-variable polynomials, the operation is only slightly different.

Definition 2.9 (Discriminant)**.** Let

$$p \coloneqq a_n \cdot x^n + a_{n-1} \cdot x^{n-1} + \ldots + a_1 \cdot x^1 + a_0 \in \mathbb{Z}[\boldsymbol{x}_1^n], \quad a_n \neq 0$$

be a polynomial of $\mathbb{Z}[\boldsymbol{x}_1^{n-1}][x]$, and let

$$p' \coloneqq n \cdot a_n \cdot x^{n-1} + (n-1) \cdot a_{n-1} \cdot x^{n-2} + \ldots + a_1$$

be the derivative of p with respect to x. Then we call

$$\underbrace{a_n \cdot}_{*} \text{discr}(p) \coloneqq \underbrace{(-1)^{n(n-1)/2}}_{*} \cdot \text{res}_x(p, p').$$

the discriminant of p [McC84, p. 30f].
However, in practice we use

$$\text{discr}(p) \coloneqq \text{res}_x(p, p').$$

without the factors marked with *, because we are only interested in doing root computations of the form

$$\text{discr}(p) \stackrel{!}{=} 0.$$

Furthermore, $\text{discr}(p)$ is a polynomial of $\mathbb{Z}[\boldsymbol{x}_1^{n-1}]$ [McC88, Thm. 3.0].

To compute the discriminant of a multi-variable polynomial, we need the resultant operation to come. Since the discriminant is itself a polynomial $\text{discr}(p)(x_1, \ldots, x_{n-1})$ of $\mathbb{Z}[\boldsymbol{x}_1^{n-1}]$ and not necessarily the zero-polynomial, whether it becomes zero depends on the specific values $\boldsymbol{\alpha}_1^{n-1}$ for variables $\boldsymbol{x}_1^{n-1}$, that is, we check if

$$\text{discr}(p)(\alpha_1, \ldots, \alpha_{n-1}) = 0.$$

Thus, the specific values determine—when we plug them in—whether the resulting 1-variable polynomial $p(\boldsymbol{\alpha}_1^{n-1}, x_n)$ of $\mathbb{Z}[x_n]$ has two collapsing roots along the x_n axis.

The resultant is our third and last polynomial operation and an indicator whether and where two polynomials have a common root. It is strongly related to the "Greatest Common Divisor" (GCD) operation for integers and polynomials. For example, the GCD of the two integers

$$30 = 2 \cdot 3 \cdot 5 \qquad \text{and} \qquad 42 = 2 \cdot 3 \cdot 7 \tag{7}$$

is the integer

$$\gcd(30, 42) = 2 \cdot 3 = 6.$$

The GCD of the two polynomials

$$p \coloneqq x^2 - 1 = (x+1) \cdot (x-1) \qquad \text{and} \qquad q \coloneqq x^2 + 3x + 2 = (x+1) \cdot (x+2) \tag{8}$$

is the non-constant polynomial

$$\gcd(p, q) = (x + 1).$$

We can check whether two integers a and b have a common (integer) divisor by computing their GCD and check if it is 1 or not, i.e., whether $gcd(a, b) = 1$ or not. We can check whether two polynomials p and q have a common (non-constant polynomial) divisor by computing their GCD and check whether $gcd(p, q)$ is a constant-polynomial or not. In that sense the GCD operation computes the GCD and is an indicator for common divisors at the same time. In contrast, the resultant of two polynomials is only an indicator for a common (polynomial) divisor without computing the GCD. The important difference for us and a CAD-algorithm is that the resultant eliminates a variable.

Definition 2.10 (Resultant)**.** Assume that we have two polynomials

$$p \coloneqq a_n \cdot x^n + \ldots + a_1 \cdot x^1 + a_0 \qquad \text{and} \qquad q \coloneqq b_m \cdot x^m + \ldots + b_1 \cdot x^1 + b_0$$

of $\mathbb{Z}[\boldsymbol{x}_1^{n-1}][x]$ with $n = deg(p) \geq 1$, $m = deg(q) \geq 1$. Let the Sylvester

Matrix [Coh03, p. 267] of p and q over variable x be

$$S(p,q,x) = \begin{array}{c} 1 \\ 2 \\ \vdots \\ m \\ 1 \\ 2 \\ \vdots \\ n \end{array} \begin{pmatrix} a_n & a_{n-1} & \dots & a_0 & & & \\ & a_n & a_{n-1} & \dots & a_0 & & \\ & \ddots & & & \ddots & & \\ & & a_n & a_{n-1} & \dots & a_0 & \\ b_m & b_{m-1} & \dots & & b_0 & & \\ & b_m & b_{m-1} & \dots & & b_0 & \\ & & \ddots & & & \ddots & \\ & & & b_m & b_{m-1} & \dots & b_0 \end{pmatrix}.$$

Then the resultant is defined as

$$\mathrm{res}_x(p,q) \coloneqq \det(S(p,q,x))$$

Furthermore, $S(p,q,x) \in D^{(m+n)\times(m+n)}$ and $res_x(p,q)$ is a polynomial of $\mathbb{Z}[\boldsymbol{x}_1^{n-1}]$.

So, we use the resultant to indicate whether two polynomials p and q have a common (non-constant polynomial) divisor. If they have such a divisor and this divisor has roots, then we know that p and q have this root in common. Note that the divisor may be a polynomial like x^2+1 that has no real roots, thus p and q may not have a common root, even though they have a common divisor. Therefore the resultant is an over-approximation of common roots, which is not optimal but acceptable in CAD.

The definition of the resultant through the determinant of a matrix with that specific coefficient arrangement is a concise, convenient notation for the resultant polynomial. Conceptually, in order to compute the resultant we construct the Sylvester matrix and compute the determinant of it. In the Sylvester matrix we copy and shift the coefficients of polynomial p, which are polynomials themselves, into subsequent rows as often as the degree m of the other polynomial q. Then we copy and shift the coefficients of q into subsequent rows as often as the degree n of p. We end up with a square matrix with $m+n$ rows and columns that we can compute the determinant for. For example, using the two single-variable polynomials

$$p \coloneqq 1x^2 + 2x + 3 \qquad \text{and} \qquad q \coloneqq 4x^3 + 5x^2 + 6x + 7,$$

with degree 2 and 3, the Sylvester matrix $S(p,q,x)$ is the 5×5 matrix

$$\begin{bmatrix} 1 & 2 & 3 & 0 & 0 \\ 0 & 1 & 2 & 3 & 0 \\ 0 & 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 & 0 \\ 0 & 4 & 5 & 6 & 0 \end{bmatrix}.$$

See [Coh03, p. 265ff.] for an understandable derivation of the resultant operation, especially to see why we arrange the coefficients in that particular way and why we then take the determinant.

In general the resultant is defined for arbitrary non-constant polynomials of $\mathbb{Z}[\boldsymbol{x}_1^{n-1}][x_n]$, and indicates that two polynomials have a common (non-constant polynomial) divisor with respect to x_n if and only if the resultant becomes zero. If p and q are polynomials of $\mathbb{Z}[\boldsymbol{x}_1^n]$, then the resultant itself is a polynomial of $\mathbb{Z}[\boldsymbol{x}_1^{n-1}]$—not necessarily the zero-polynomial. So, like the discriminant the resultant is actually a function of variables $\boldsymbol{x}_1^{n-1}$ and whether it becomes zero depends on the specific values for these variables, that is,

$$(\mathrm{res}_{x_n}(p,q))(x_1,\ldots,x_{n-1}) \stackrel{!}{=} 0.$$

In other words, the resulting constant number

$$(\mathrm{res}_{x_n}(p,q))(\alpha,\cdots,\alpha_{n-1}),$$

after plugging in specific values $\boldsymbol{\alpha}_1^{n-1}$ for the variables $\boldsymbol{x}_1^{n-1}$, tells us whether the resulting single-variable polynomials $p(\boldsymbol{\alpha}_1^{n-1}, x_n)$ and $q(\boldsymbol{\alpha}_1^{n-1}, x_n)$ of $\mathbb{Z}[x_n]$ have a common (non-constant polynomial) divisor in x_n. If they do, they potentially have a common root along the x_n axis.

In practice, the resultant is not computed as in Def. 2.10, because there are more efficient algorithms [Duc00].

While computing the coefficients with respect to a certain variable did not depend on any specific property of an algebraic structure, this is different for computing the discriminant and the resultant. Actually, these only work for polynomials $U[x]$, where U is a "Unique Factorization Domain" (UFD), a slightly more concrete, mathematical structure than a ring.

As a quick reminder: The relation between the more widely known mathematical structures fields and rings, and the less widely known UFDs is as follows:

$$\text{rings} \supset \text{UFDs} \supset \text{fields}.$$

Thus, every field is automatically a UFD and every UFD is a ring. On the other hand, a ring is a generalization of a UFD, and a UFD is generalization of a field, so if we prove some property for rings, then these automatically hold for UFDs and fields as well. See Geddes et al. [GCL92] for the precise definitions.

Lemma 2.1 (Polynomial Ring)**.** Let F be a field. Then the set of single-variable polynomials $F[x]$ is a ring.

It is a well-known fact from Lemma 2.1 that the polynomials $\mathbb{Q}[x]$ with rational coefficients form a ring, because $\mathbb{Q}$ is a field. However, it's less well-known why the polynomials $\mathbb{Z}[x]$ form a ring, since $\mathbb{Z}$ is not a field.

Lemma 2.2 (Unique Factorization Domain)**.** Let U be a UFD. Then the set of single-variable polynomials $U[x]$ is a UFD.

Instead the set $\mathbb{Z}$ of integers is a UFD, which by Lemma 2.2 carries over into $\mathbb{Z}[x]$, which also makes it a ring due to rings $\supset$ UFDs. This is why the desired discriminant and resultant operations are well-defined for the single-variable polynomials of $\mathbb{Z}[x]$.

Corollary 2.1. $\mathbb{Z}[\boldsymbol{x_1^{n-1}}]$ is a UFD.

Proof 2.1. Since $\mathbb{Z}$ is a UFD, this follows by induction by repeatedly applying Lemma 2.2 on

$$((((\mathbb{Z}[x_1])[x_2])\ldots)[x_{n-2}])[x_{n-1}].$$

From Corollary 2.1 it follows that the discriminant and resultant operations are well-defined for polynomials of $\mathbb{Z}[\boldsymbol{x_1^n}]$ as well.

For the following operations, it is necessary to define the notion of a "prime" polynomial. Just as any integer can be decomposed into its prime factors [Coh03, Thm.2.17,p. 27] as in Eq. 7, which is unique up to the order of the factors, any polynomial of $\mathbb{Z}[\boldsymbol{x_1^n}]$ can be decomposed into its unique "prime" polynomial factors [Coh03, Thm.4.40,p. 139,205], called irreducible polynomials, as in Eq. 8. This is one of the characterisitic properties of $\mathbb{Z}[\boldsymbol{x_1^n}]$ being a UFD.

Definition 2.11 (Irreducible Polynomial)**.** Let $p, q, r \in \mathbb{Z}[\boldsymbol{x_1^n}]$. A polynomial p is called reducible if there are non-constant polynomials q and r such that

$$p = q \cdot r.$$

A polynomial p is called irreducible if there are no such q and r [Coh03, p. 118,204].

Definition 2.12 (Irreducible Decomposition of Polynomials)**.** Given a polynomial p and a polyset P of $\mathbb{Z}[\boldsymbol{x}_1^n]$, let

$$\text{irred}(p) \coloneqq \{p_1, \ldots, p_k\}$$

be the irreducible decomposition of p with $p_1, \ldots, p_k \in \mathbb{Z}[\boldsymbol{x}_1^n]$ being irreducible polynomials such that

$$p = c \cdot p_1 \cdot p_2 \ldots p_k \text{ for some } c \in \mathbb{Z}$$

and let

$$\text{irred}(P) \coloneqq \bigcup_{p \in P} \text{irred}(p)$$

be the elementwise decomposition of the polyset P.

2.3.2 The McCallum-Brown CAD projection

Having defined the notion of a irreducible polynomial, polynomial level as well as the leading coefficient, discriminant and the resultant operations, we can define the the projection operator, called "projector", that the McCallum-CAD algorithm uses in the "projection" phase.

Definition 2.13 (McCallum-Brown-Full-Projector [BK15])**.** Assume we have a finite set P of irreducible polynomials of $\mathbb{Z}[\boldsymbol{x_1^n}]$, each of the same level k with $1 \leq k \leq n$ and let

$$\text{leadCoeff}(P) \coloneqq \bigcup_{p \in P} \text{leadCoeff}_{x_k}(p), \qquad \text{res}(P) \coloneqq \bigcup_{p,q \in P, p \neq q} \text{res}_{x_k}(p,q),$$

$$\text{discr}(P) \coloneqq \bigcup_{\substack{p \in P \\ deg_{x_k}(p) \geq 2}} \text{discr}_{x_k}(p),$$

then the set of polynomials

$$\mathbb{P}(P) \quad \coloneqq \quad \text{leadCoeff}(P) \quad \cup \quad \text{discr}(P) \quad \cup \quad \text{res}(P),$$

is called the "projection of P." It contains only polynomials of at most level $k-1$.

The metaphor of a "projector" or "projection-step" used in the name of the operation in Def. 5.2 is accurate, because $\mathbb{P}(P)$ only contains polynomials of a smaller level, i.e., with fewer variables for a lower-dimensional space, just as the shadow on the ground surface of a stick in 3-dimensional space is a lower, 2-dimensional projection.

We only compute the discriminant for polynomials that have a degree of at least 2 in the variable x_k, because otherwise there are no "double roots", which the discriminant is computed for. Additionally, this projector is defined for irreducible polynomials of the same level only. This restriction is not absolutely necessary, but it eases the presentation and understanding compared to presentations where the projector is defined for polynomials of any level.

Finally, the name to describe the projector in Def. 5.2 is used to distinguish it from other projection variants. Since there are multiple variants of projections, there are two aspects, foundation and cell coverage, to take into account when we compare projection operators:

First, there is a coarse separation of projection operators in their theoretical foundation. A foundation prescribes what polynomials have to be computed to construct a valid cylindric decomposition. In Def. 5.2 we use Scott McCallum's foundation [McC84] with improvements by Brown [Bro01]: Brown proved that computing leading coefficients, discriminants and pairwise resultants are sufficient for a CAD. The other famous foundation by Collins is compared in Sec. 5.1.

The second aspect to consider for the projection is the cell coverage. Normally we want to compute a "full" decomposition into multiple non-overlapping CAD cells that cover the whole space as in Def. 5.2. However, there are also "single cell" or a "partial" CAD variants that vary in the way of projection or lifting. Since a "single cell" variant , as covered in Sec. 2.5, is the main topic of this thesis, we will see how such a variant called "OneCell" modifies the CAD projection by Brown. Most notably, it reduces set of polynomials that we need to compute.

Let us take a closer look on what the projector in Def. 5.2 does by looking analytically at the example we have previously looked at intuitively in Sect. 2.1.4

about decomposing the real x-y-space. Recall that we want to construct x-cells which we extend into cylinders along the y-axis with the property that no matter which x-value we chose within an x-cell, the number of roots and their ordering along the y-axis stays the same, because this is the necessary property to extend the x-cells into x-y-cells. And as it turns out, the coefficients, the discriminants and resultants of x-y polynomials are the x-polynomials whose roots tell us which x-cells to construct.

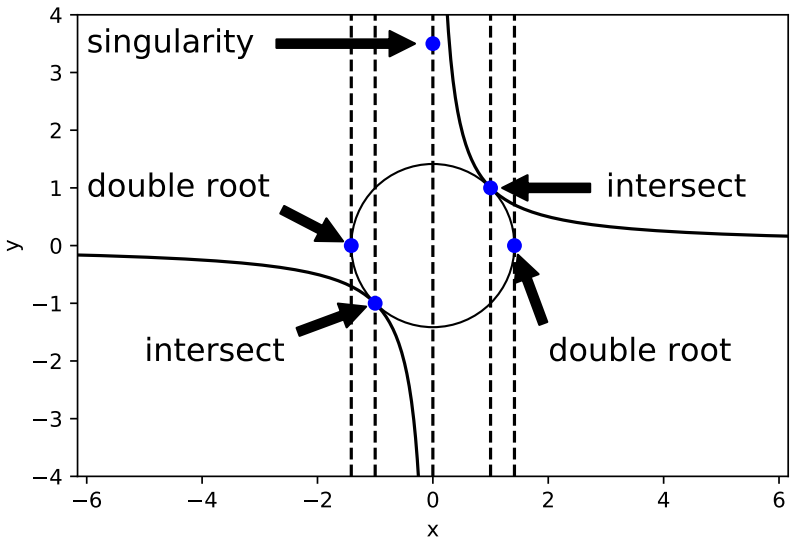


Figure 2.14: Root plot and special points—double roots, singularities and intersections—of polynomials $x^2 + y^2 - 2 \stackrel{!}{=} 0$ and $xy - 1 \stackrel{!}{=} 0$.

We start with an inital set of two x-y-polynomials

$$p \coloneqq x^2 + y^2 - 2 \qquad \text{and} \qquad q \coloneqq xy - 1$$

of $\mathbb{Z}[x, y]$, which we reintpret as the $\mathbb{Z}[x][y]$ polynomials

$$p \coloneqq (1) \cdot y^2 + (0) \cdot y + (x^2 - 2) \qquad \text{and} \qquad q \coloneqq (x) \cdot y + (-1)$$

whose roots

$$p \stackrel{!}{=} 0 \qquad \text{and} \qquad q \stackrel{!}{=} 0$$

are visualized in the root plot in Fig. 2.14.

First, we compute the leading coefficients with respect to variable y

$$\text{coeffs}(\{p, q\}) = \{1, x\}.$$

These are the x-polynomials whose roots tell us the x-coordinate of a singularity on the x-y-plane and where we thefore have to create a separate x-cell. For example, the leading coefficient x of polynomial q has the root 0 and therefore tells us that there is singularity at $x = 0$.

Second, we compute the discriminants with respect to variable y

$$\text{discr}(\{p, q\}) = \{x^2 - 2\}.$$

These discriminants are the x-polynomials whose roots tell us the x-coordinate of a point on the x-y-plane where two roots along the y-axis collapse into one—we call this a double root. There again we have to create a separate x-cell. In this example we only compute the discriminant for p, because q has only degree 1 in variable y and therefore has no possibility of a double root. along y. This discriminant $x^2 - 2$ of p has the roots $-\sqrt{2}$ and $\sqrt{2}$, which precisely correspond to the x-coordinates of the double root points of the circle in Fig. 2.14. At each of these points two roots, which exist for $x \in (-\sqrt{2}, \sqrt{2})$ in the upper and lower circle halfs, collapse.

Third, we compute the resultant with respect to variable y between any two polynomials of our set.

$$\text{res}(\{p, q\}) = \{(x + 1)^2 \cdot (x - 1)^2\}$$

The resultants are the x-polynomials whose roots tell us the x-coordinate of an intersection between two x-y-polynomials and where we have to create a separate x-cell. Since we only have two polynomials, there is just one resultant to compute. This resultant has the roots -1 and 1 which corresponds precisely to the x-values of the left and right intersection points of the circle with the two curves in Fig. 2.14.

The McCallum-Brown projector in Def. 5.2 alone is not sufficient to create a CAD; it has to be applied repeatedly.

Definition 2.14 (McCallum-Brown-Full-Projection)**.** Given a polyset P of $\mathbb{Z}[\boldsymbol{x}_1^n]$ of arbitrary mixed levels, categorize the irreducible decomposition of its polynomials

$$\text{irred}(P) \stackrel{!}{=} P_n \cup P_{n-1} \cup \ldots \cup P_1 \cup P_0$$

into buckets $P_n, \ldots P_0$ where bucket P_i contains and only contains the polynomials of level i. Then for $i = n, n-1, \ldots, 2$ in that order compute after Def. 5.2 and Def. 2.12

$$\text{irred}(\mathbb{P}(P_i))$$

and add/categorize the resulting polynomials into buckets $P_i, P_{i-1}, \ldots, P_0$. The final set of resulting buckets $P_n, \ldots, P_1$ without P_0 is called

$$\text{PROJ}(P),$$

the "McCallum-Brown-Full-Projection of P".

In this thesis we use a definition adapted from the projector given by Brown and Košta [BK15], which applies the projector only to polynomials of the same level.

2.3.3 Another view on projection

The CAD projection as in Def. 2.14, does an irreducible decomposition initially and after each projection step. Thereby it ensures that a projector only has to deal with irreducible polynomials of the same level, which simplifies the projector operation.

This projection applies the McCallum-Brown-Full-Projector (Def. 5.2) once on each "bucket". A bucket is the current collection of all polynomials of that level, and categorizes the resulting polynomials into the lower-level buckets. It is useful to visualize this filling of lower-level buckets to get a better understanding of the single-cell variant in Sec. 2.5 and how it differs from this projection. In Fig. 2.15 we see that initially we start with polynomials p, q, r, s, t on several

$\rightarrow P_3$:	p	q	
P_2:	r	$\text{discr}(p)$	$\text{leadCoeff}(q)$
P_1:	s	$\text{discr}(q)$	$\text{res}(p, q)$
P_0:	t	$\text{leadCoeff}(p)$	

Figure 2.15: Qualititative initial categorization of input polynomials p, q, r, s, t into level buckets as well as a categorization of polynomials from the first projection-step.

levels, the highest level being 3. Thus, the first projector operation works on the bucket of polynomials of the highest level n, P_3 in this case, and creates polynomials of a smaller level which are categorized accordingly. For example, the discriminant of p, $\text{discr}_{x_3}(p)$ may end up in P_2, the leading coefficient $\text{leadCoeff}(p)$ may end up in P_0, and the resultant $\text{res}_{x_3}(p, q)$ may end up in P_1. Normally, these polynomial operations take a variable as a subscript. Here, for simplicity we assume that they work on the correct, third variable for some variable order. In order to further simplify the visualization we assume that the initial and all resulting polynomials are irreducible, which is not necessarily the case in general. In general, a single resulting polynomial like $\text{discr}(p)$ may be split into several irreducible polynomials. The next projection operation works

P_3:	p	q			
$\rightarrow P_2$:	r	$\text{discr}(p)$	$\text{leadCoeff}(q)$		
P_1:	s	$\text{discr}(q)$	$\text{res}(p, q)$	$\text{discr}(r)$	$\text{discr}(\text{discr}(p)) \ldots$
P_0:	t	$\text{leadCoeff}(p)$	$\text{leadCoeff}(r)$	$\text{res}(r, \text{discr}(p))$	$\ldots$

Figure 2.16: Categorization of polynomials into level buckets (continued) after the second projection-step.

on the bucket of polynomials of level $n-1$, P_2 in this case, and again creates polynomials of smaller levels, which are categorized accordingly. All the resulting polynomials like $\text{discr}_{x_3}(p)$ from the previous step are being treated as if

they belonged to the initial set of polynomials. This is why we also compute polynomials like

$$\text{leadCoeff}_{x_2}(\text{discr}_{x_3}(p)), \qquad \text{discr}_{x_2}(\text{discr}_{x_3}(p)), \qquad \text{res}_{x_2}(r, \text{discr}_{x_3}(p)).$$

This process is repeated until the last bucket P_2 is projected. Afterwards the bucket P_0 is discarded, because we are interested in roots, which the constant-polynomials of P_0 do not have.

Summarizing, in constrast to the single-level projector from Def. 5.2, the full projection from Def. 2.14 ensures that the polynomials of even the highest levels are projected down to bucket P_1 and are taken into account when we create the initial, lowest-dimensional cells.

2.3.4 Projection factor set

Since there is no uniform vocabulary in the literature to separate the repeated application of a projector from a single application, we call the repeated application—on all levels of polynomials—a "projection", and the single application—on a single level of polynomials—a "projection step".

Furthermore and to the best of our knowledge, Brown [Bro01, p. 448] introduced the term "projection factor set" to refer to all polynomials that are the result of the projection phase.

Definition 2.15 (Projection factor and projection factor set)**.** Let $P_n, \ldots, P_1$ be the final set of buckets (without P_0) of the projection of Def. 2.14

$$\text{PROJ}(P).$$

Then the union of those buckets

$$\bigcup_{i=1}^{n} P_i$$

is called the *projection factor set* and every polynomial in that set is a *projection factor*.

The projection factor set by definition (see Def. 2.15) does not necessarily include in the initial input polynomials, but their irreducible factorizations.

This projection factor set then "provides an implicit representation of the CAD" and the subsequent phases converts it in "an explicit representation of this CAD" [Bro01, p. 447].

2.4 Lifting Phase for a full CAD

The lifting-phase, as the name suggests, deals with "lifting" a level-$k-1$ cell into a level-k cell, that is, creating a higher-dimensional cell from a lower-dimensional one. The end result of the lifting-phase is a decomposition of whole universe into cells of the highest level, that is, of all dimensions. This phase is actually known under "extension" or "lifting" depending on whether we look at the cell construction geometrically or algebraically. If we look geometrically then the word "extending" is appropiate, because we extend a cell along a new axis. If we

look algebraically, then the term "lifting" is more appropiate, because we "lift" a level-$k-1$ cell—using polynomials of level-$k-1$ and lower—by one level into a level-k cell. These polynomials come from the buckets that are created during the projection-phase (see Sec. 2.3). Each bucket contains all polynomials of a certain level and these buckets are hierarchically created sorted top-down from highest to lowest level. In contrast, during the lifting-phase we work through these buckets from the bottom to the top so that this bottom-up metaphor of lifting is more appropriate.

"Lifting" in CAD starts with a "base case". The "base case" deals with level-1 bucket P_1, which contains all 1-variable polinomials of the projection factor set (see Def. 2.15, which is the final result of the projection phase (see Sec. 2.3. The variable in all those polynomials, called x_1 without loss of generality, is the first one in the variable order

$$x_1 < x_2 < \ldots < x_n$$

and the last one in the order in which the projection-phase eliminates variables from polynomials. This variable however is not eliminated itself.

2.4.1 Basic root isolation

As we intuitively described in Sec. 2.1.3, we compute all the roots of all polynomials in P_1 to construct what we call level-1 CAD-cells or x_1-cells. This process of computing a root or all the roots is called "root isolation". The level-1 cells are the basic, 1-dimensional cells that are either closed single-point intervals of a root, or open intervals between two consecutive roots. Let

$$r_1, \ldots, r_k \in \mathbb{R} \text{ with } r_i < r_{i+1},\ i = 1, \ldots, k-1$$

be the pairwise distinct, increasingly-ordered roots of polynomials in P_1. Then

$$\{(-\infty, r_1), \quad [r_1, r_1], \quad (r_1, r_2), \quad [r_2, r_2], \ldots, [r_k, r_k], \quad (r_k, \infty)\}$$

are the x_1-CAD-cells. These cells are sign-invariant on the polynomials in P_1 by construction. For example, if we have the bucket

$$P_1 := \{x_1^2 - 1, \quad 2x_1\}, \tag{9}$$

then the roots of these polynomials are

$$\{-1, 0, 1\},$$

and we construct the following x_1-cells:

$$\{\underbrace{(-\infty, -1)}_{\text{sector}}, \quad \underbrace{[-1, -1]}_{\text{section}}, \quad \underbrace{(-1, 0)}_{\text{sector}}, \quad \underbrace{[0, 0]}_{\text{section}}, \quad \underbrace{(0, 1)}_{\text{sector}}, \quad \underbrace{[1, 1]}_{\text{section}}, \quad \underbrace{(1, +\infty)}_{\text{sector}}\}$$

We call those closed invervals a "section" and those open intervals a "sector", and represent them formally as (in)equalities:

$$x_1 = constant \qquad \text{(section)}$$

or

$$constant < x_1 < constant \qquad \text{(sector)}$$

As we move into higher dimensions, a cell will be represented by sequence of sections and sectors, which is why we also call these "cell components".

2.4.2 Root expressions for real algebraic numbers

In CAD one of the vital operations on polynomials is root isolation of 1-variable polynomials. However, a real root cannot always be adequately represented in finite memory, since it may have infinitely many decimal places, and approximating it with a float, rational or any other finite-precision representation may introduce rouding errors, which we cannot accept.

Instead, we represent potentially problematic real numbers such as roots implicitly by a so-called "root-expression".

Definition 2.16 (Single-variable Root-Expression)**.** Given an integral, single-variable polynomial $p(x) \in \mathbb{Z}[x]$ with real roots

$$r_1 < r_2 < \ldots < r_k \text{ with } 1 \leq k \leq deg_x(p)$$

and an integer idx $\in \{1, \ldots, k\}$, we call the pair

$$\text{root}(p(x), \text{idx}) := r_{\text{idx}} \in \mathbb{R}$$

a *root-expression* to represent a specific, isolated real root value.

A root-expression as in Def. 2.16 is a pair containing a polynomial and a so-called "root number" or "root index". The index refers to one of the polynomial's roots: We order all the roots from smallest to highest and the first one gets index 1, the next one gets index 2 and so on. If we need the real number explicitly, we can compute it on demand and up to an appropiate decimal place.

A real number for which such an implict representation exists, is called an "algebraic real".

Definition 2.17 (Real algebraic numbers)**.** A real number $r \in \mathbb{R}$ is called algebraic if and only if there is a $p(x) \in \mathbb{Z}[x]$ such that

$$p(r) = 0,$$

that is, if it's the root of some single-variable polynomial with integer coefficients.

The set of all real algebraic numbers—often abbreviated to "algebraic real" or "RAN"—is $\mathbb{R}_{alg} \subsetneq \mathbb{R}$ is a proper subset of the reals. They are called algebraic, because they can be represented as roots of integral, 1-variable polynomials.For example, the real number $\sqrt{2}$ is algebraic, because it is a root of a polynomial $x^2 - 2$: It can be represented by the root-expression

$$\text{root}(x^2 - 2, 2),$$

because $\sqrt{2}$ is the second root of the polynomial $x^2 - 2$. However, there is no polynomial with integer coefficients whose roots include π or e; these numbers belong to the complement set of "transcendental numbers".

Alternatively and in order to avoid enumerating roots, we can replace a root-index with an "isolating interval". This interval encloses a single root and its

bounds are rational numbers with a finite representation. So, $\sqrt{2}$ can also be represented by

$$(x^2 - 2, [\frac{7}{5}, \frac{8}{5}]),$$

because it's the only root of this polynomial within the interval $[\frac{7}{5}, \frac{8}{5}] = [1.4, 1.6]$.

Summarizing, there is always an implicit, finite, representation for algebraic reals. One huge advantage is that $\mathbb{R}_{alg}$ is closed under arithmetic, that is, it's possible to add and multiply algbraic reals in their implicit form and always get another algebraic real in an implicit form back—without rounding errors. This is invaluable when we need to evaluate polynomials, as we will need for root isolation with multi-variable polynomials later on. As such, for any practical purposes we work on $\mathbb{R}_{alg}$ instead of $\mathbb{R}$.

Coming back to CAD: Because the boundaries of a 1-dimensional cell are real numbers that come from isolating roots—potentially problematic—, we would first create the cell $[-1, -1]$—constructed from the polynomial $x_1^2 - 1$ in Eq. 9—as the section

$$x_1 = \underbrace{\text{root}(z^2 - 1, 1)}_{\to 1} \qquad \text{(section)},$$

because the first root of $z^2 - 1$ is -1. Afterwards we would simplify it to

$$x_1 = 1 \qquad \text{(section)}.$$

In the same manner we would first create the cell $(0, 1)$—constructed from the $x^2 - 1$ and $2x_1$ in Eq. 9—as the sector

$$\underbrace{\text{root}(2z, 1)}_{\to 0} < x_1 < \underbrace{\text{root}(z^2 - 1, 2)}_{\to 1} \qquad \text{(sector)},$$

because the first and only root of $2z$ is 0 and the second root of $z^2 - 1$ is 1. Afterwards we would simplify it to

$$0 < x_1 < 1 \qquad \text{(sector)}.$$

Notice that we replaced the variable x_1 with z in the root-expressions. The choice of z is arbitrary—it just shouldn't appear anywhere else—and by convention this is done to highlight that z in the root-expression has nothing to do with x_1 anymore. In practice, we

2.4.3 Delineability of lower-level cells

Before we present the actual lifting procedure, it is useful to know what property lifted cells have to satisfy. The main property that we want to have satisfied by the final constructed cells is this: Each cell must be sign-invariant for the initial input polynomials; for each input polynomial every point in a cell must produce the same sign, when plugged into the polynomial.

However, before we start to explain the lifting formally, we need the definition of *delineability*, that will ensure, that lifting a cylindric level-k-1 cell in to several cylindric level-k cells is possible, and the formal definition of a section and sector, which will allow us to algebraically represent a cell.

For the definition of "delineability" we need the notion of a cylinder. We use the definition of a cylinder from [McC84, p.50].

Definition 2.18 (Cylinder over lower-dimensional space)**.** Given a connected space $S \subseteq \mathbb{R}^{n-1}$, we call

$$Z(S) := S \times \mathbb{R}$$

the cylinder over S.

A cylinder as defined in Def. 2.18 takes a given level-$n-1$ CAD-cell, involving only the first $n-1$ axes, and stretches it into positive and negative infinity along a new, n-th axis as we have intuitively described in Fig. 2.9 from Sec. 2.1.4.

We use the definition of delineability of a function from McCallum's Phd thesis [McC84, p.37] that is restated in Brown's improvement paper [Bro01, p.448]:

Definition 2.19 (Delineability of a function over lower-dimensional space)**.** A real polynomial $p \in \mathbb{Z}[\boldsymbol{x}_1^n]$ is delineable over a space $S \subseteq \mathbb{R}^{n-1}$ if the two following properties hold:

1. The set of real roots of p lying in the cylinder over S can be defined by the range of k continuous real functions

   $$f_1, f_2, \ldots, f_k \colon S \to \mathbb{R},$$

   which map from S to $\mathbb{R}$, with $k \geq 0$ and

   $$\forall \boldsymbol{\alpha}_1^{n-1} \in S : \; f_1(\boldsymbol{\alpha}_1^{n-1}) < f_2(\boldsymbol{\alpha}_1^{n-1}) < \ldots < f_k(\boldsymbol{\alpha}_1^{n-1}),$$

   where point $\boldsymbol{\alpha}_1^{n-1} := \alpha_1, \ldots, \alpha_{n-1}$.

2. For each f_i there is an integer $m_i \geq 1$ such that we have the same multiplicity m_i of the root $f_i(\boldsymbol{\alpha}_1^{n-1})$ at $p(\boldsymbol{\alpha}_1^{n-1}, x_n)$ for every point $\boldsymbol{\alpha}_1^{n-1} \in S$.

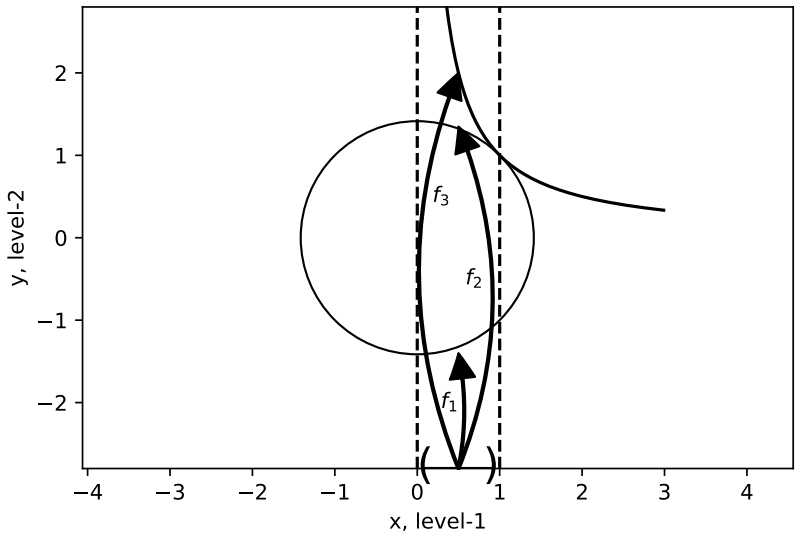


Figure 2.17: Qualitative root plot of level-2 polynomials $x^2 + y^2 - 2$ and $xy - 1$, that are delineable over a level-1 cell $(0, 1)$. We see a cylinder over this cell and three continuous real functions f_1, f_2, f_3 that map the points of the cell to the real roots of the polynomials lying in the cylinder.

The definition of delineability in Def. 2.19 is a formal way to state the conditions under which we will be able cut the cylinder over a level-$n-1$ CAD-cell S (formally $S \times \mathbb{R}$) into a stack of level-n cells. It describes a relationship between this level-$n-1$ cell S and a level-n polynomial p that makes this construction possible. It states that there must exist functions, called f_i which map from the cell S to the roots of p inside the cylinder over S. These functions are not allowed to overlap in their range. It also states that for each function f_i all the points in its range, correspond to roots of p with the same multiplicity. Another way to phrase these delineable conditions is that no matter which level-$n-1$ point we choose from the cell, the (1-variable) poly p at that point has the same number of roots in the same order and each root has the same multiplicity.

A qualititative example can be found in Fig. 2.17. There we have a level-$k-1$ cell and the functions f_1 to f_3, which map points of this cell to the roots of a polynomial without overlap; actually, we look at the roots of two delineable polynomials, but this doesn't change the the first property of Def. 2.19. For example, if we look at the segment of all the points the function f_1 maps to—this set of point is called the range of f_1—, then the multiplicity of the root of the polynomial at each of these points is 1. The same holds true for functions f_2 and f_3, and refers to the second property of Def. 2.19.

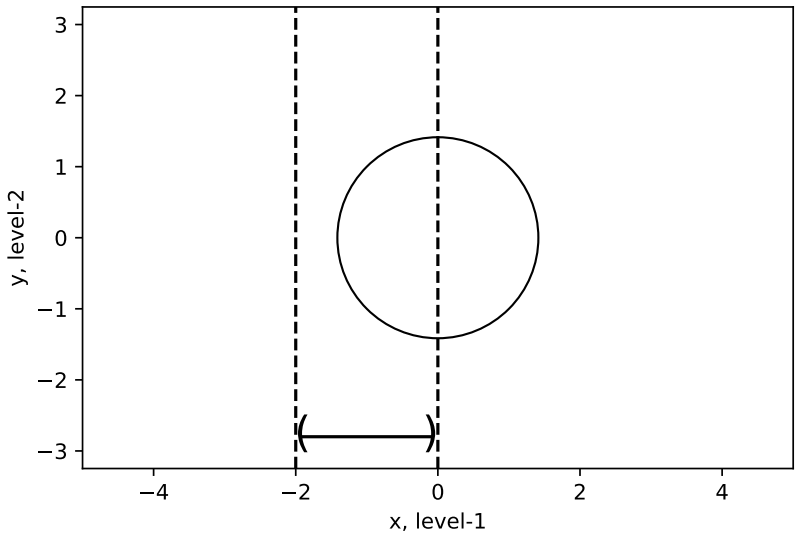


Figure 2.18: Qualitative root plot of level-2 polynomial $x^2 + y^2 - 2$, that is not delineable over a level-1 cell $(-2, 0)$.

An example for a polynomial that is not delineable on a cell can be found in Fig. 2.18. There cannot exist any real, continuous function to map the points within the level-1 cell to the roots of the polynomial, because in the interval $(-2, -\sqrt{2})$ on the x-axis there are no roots along the y-axis to point to. Also, the multiplicity of the root at coordinate $x = -\sqrt{2}$ is 2—a double root—whereas the roots with an x-coordinate within $(-\sqrt{2}, 0)$ would have a multiplicity of 1. Furtheremore, the required functions f_1 and f_2 would overlap at x-coordinate $-\sqrt{2}$.

2.4.4 Sections and sectors analytically

The notion of delineability lets us define—now precisely—what sections and sectors [McC84, p.37] are:

Definition 2.20 (Section/Sector over a space)**.** Given a real polynomial $p \in \mathbb{Z}[\boldsymbol{x}_1^n]$ that is delineable over a space $S \subseteq \mathbb{R}^{n-1}$, then for each f_i of the k defining, continuous functions from Def. 2.19

$$f_1, f_2, \ldots, f_k \colon S \to \mathbb{R},$$

the graph of f_i is called an p-section, or simply *section*, and the region between two consecutive sections is called a p-sector, or simply *sector*.

The definition Section in Def. 2.20 is the formal way to define what will become the boundaries of our CAD cells.

We use these functions f_i—for a section—and conscutive pairs (f_i, f_j)—for sectors—for our representation of the CAD-cells. In Fig. 2.17 we see that we can

extend the level-$k-1$ cell into an cylinder. The roots of the involved polynomials cut this cylinder into sections and the spaces inbetween become sectors. Each of these sections and sectors—called "cell components"–appended to the level-$k-1$ cell makes up a new level-k cell. So, a section is defined by a single function and a sector is defined by two—although a sector can also have $-\infty$ and $+\infty$ bounds. For example, if we start with a level-1 cell

$$[\underbrace{0 < \ x_1 \ < 1}_{\text{sector}}]$$

as in Fig. 2.17, we can formally represent the level-2 cell that corresponds to the segment at the range of f_1 with

$$[\underbrace{0 < \ x_1 \ < 1}_{\text{sector}}, \quad \underbrace{x_2 \ = f_1(x_1)}_{\text{section}}]$$

and the level-2 cell that lies in between the segments of f_1 and f_2 with

$$[\underbrace{0 < \ x_1 \ < 1}_{\text{sector}}, \quad \underbrace{f_1(x_1) < \ x_2 \ < f_2(x_1)}_{\text{sector}}].$$

Similarly, we represent the level-2 cell that lies above the segment of f_3 with

$$[\underbrace{0 < \ x_1 \ < 1}_{\text{sector}}, \quad \underbrace{f_3(x_1) < \ x_2 \ < +\infty}_{\text{sector}}].$$

An interesting trick is used to find such functions f_1 to f_3, that map points within a cell into points that are roots of polynomials: Use the polynomials themselves; they already contain an suitable—although implicit— mapping to their roots.

2.4.5 Full root expressions

In Fig. 2.17 we see that $p(x, y) := x^2 + y^2 - 2$—whose root plot is the circle—is a suitable representation for the circle segments which f_1 and f_2 are pointing to. Once we plug in and fix a value for the x variable, we get a polynomial in y with two roots. The first root—hypothetically root index 1—always lies on the lower segment of the circle and the second—hypothetically with root index 2—always lies on the upper segment.

This suggest to represent mappings like f_1 or f_2 by a so-called (multi-variable) "root-expression", an extension of the single-variable-root-expressions.

Definition 2.21 (Root-Expression)**.** Given an integral, multi-variable polynomial $p(x_1, \ldots, x_n) \in \mathbb{Z}[\boldsymbol{x}_1^n]$ and a positive integer idx $\in \{1, \ldots, deg_{x_n}(p)\}$, we call the pair

$$\text{root}(p(x_1, \ldots, x_n), \text{idx})$$

a (multi-variable) *root-expression*.
It is evaluated by plugging a point $(\alpha_1, \ldots, \alpha_{n-1}) \in \mathbb{R}^{n-1}$, into the polynomial p:

$$q(x_n) := p(\alpha_1, \ldots, \alpha_{n-1}, x_n) \in \mathbb{Z}[x_n].$$

Then the evaluation result is either the real number represented by the

single-variable root-expression (see Def. 2.16)

$$\mathrm{root}(q(x_n), \mathrm{idx}),$$

or "undef" if $q(x_n)$ has no root with index idx.

A root-expression with a single variable is used to represent a single real number—an algebraic real to be precise—while a (multi-variable) root-expression as in Def. 2.21 is used to represent a function

$$f\colon \mathbb{R}^{n-1} \to \mathbb{R}, \quad f(x_1, \ldots, x_{n-1}) \mapsto \mathrm{root}(p(x_1, \ldots, x_{n-1}, x_n), \mathrm{idx})),$$

from a point to an algebraic real. function of the first variables $x_1, \ldots, x_{n-1}$.

In the context of CAD we use a multi-variable root-expression to represent sections and the boundaries of sectors, which are also sections. This works because by construction a level-n polynomial p is delineable over any level-$n-1$ cell. If p is delineable, it has the same number of roots in the same order "on top of" every point within the level-$n-1$ cell. In Fig. 2.19 for example, the

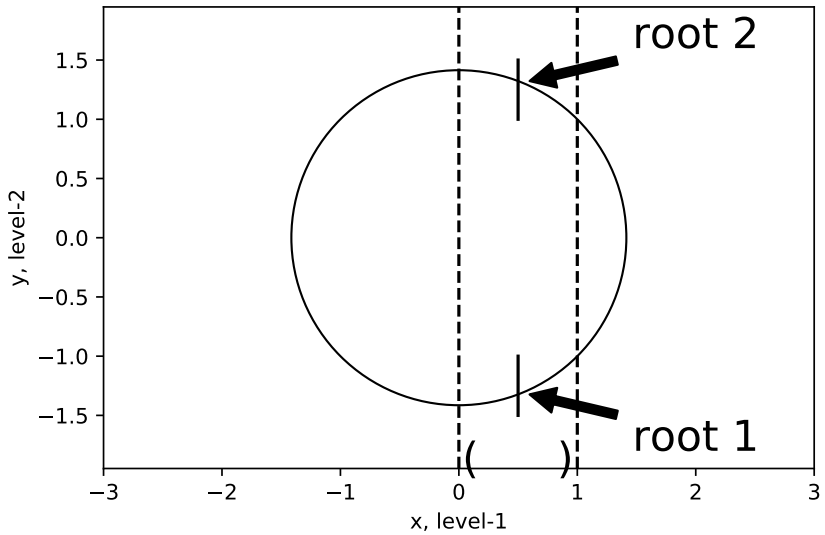


Figure 2.19: Qualitative root plot of level-2 polynomial $x^2 + y^2 - 2$, which is delineable over a level-1 cell $(0, 1)$. At coordinate $x = 0.5$ this polynomial has two roots along the level-2 y-axis.

polynomial $p(x, y) := x^2 + y^2 - 2$ has two roots in the same order on top of every point within the visualized level-1 cell $(0, 1)$. Analytically, if we plug in a point from this cell, say 0.5, we get the single-variable polynomial

$$p(0.5, y) = 0.5^2 + y^2 - 2 = y^2 - 1.75, \tag{10}$$

whose first root $-\sqrt{1.75}$—think root-index 1—lies on the lower circle segment and its second root $\sqrt{1.75}$—with root-index 2—lies on the upper segment. So,

the root-expression

$$\text{root}(p(x,y),1)$$

will refer to the lower circle segment and we can represent, for example, the level-2 cell within those two circle segments by the sequence

$$\underbrace{[\underbrace{0 < \ x \ < 1}_{\text{level-1 sector}},}_{\text{new level-2 sector}} \quad \underbrace{\underbrace{\text{root}(p(x,y),1)}_{f^{\text{low}}(x)} < \ y \ < \underbrace{\text{root}(p(x,y),2)}_{f^{\text{high}}(x)}]}_{\text{new level-2 sector}}.$$

Summarizing and ignoring the fact that we use root-expressions as implicit representation for functions, a cell is represented as a sequence of section or sector constraints, one for each variable x_i:

$$\begin{array}{rlll} constant < & x_1 & < constant & \text{(sector)} \\ f_2^{low}(x_1) < & x_2 & < f_2^{high}(x_1) & \text{(sector)} \\ & x_3 & = f_3(x_1, x_2) & \text{(section)} \\ f_4^{low}(x_1, x_2, x_3) < & x_4 & < f_4^{high}(x_1, x_2, x_3) & \text{(sector)} \\ & \dots & & \\ & x_n & = f_n(x_1, \dots, x_{n-1}) & \text{(section)} \end{array}$$

Here the (dynamic) bounds of variable x_i depend on functions of variables $x_1, x_2, \dots, x_{i-1}$.

2.4.6 A cell represented by a logical formula

As an aside: We want to use a cell as a logical formula in the context of Satisfiability Modulo Theories (SMT) (see Sec. 3), so we need to convert this cell's representation into a one.

Definition 2.22 (Defining formula)**.** Given a CAD-cell as a sequence of cell components

$$[Comp_1, Comp_2, \dots, Comp_n],$$

each a section or a sector, we call

$$\bigwedge_{i=1}^{n} Comp_i$$

the *defining formula* of the cell. If $Comp_i$ is a section, we have

$$Comp_i \coloneqq \quad x_i = f_i(x_1, \dots, x_{i-1}),$$

and if $Comp_i$ is a sector, we have

$$Comp_i \coloneqq \quad f_i^{low}(x_1, \dots, x_{i-1}) < x_i \quad \wedge \quad x_i < \quad f_i^{high}(x_1, \dots, x_{i-1})$$

Fortunately, the translation is quite simple: We just have to separate out each (in-)equality and combine these with the logical AND-operator $\wedge$, expressing that all these constraints have to hold simultaneously.

2.4.7 A single lifting-step

A "lifting-step" creates the next section or sector components of a cell. In other words, if we have a level-$k-1$ cell, which has constraints for variables $x_1, \ldots, x_{k-1}$, we create the constraint for the next variable x_k.

In order to define lifting formally, we first need the notion of a lifting-basis, a term introduced by Brown [Bro05, p. 4].

Definition 2.23 (Lifting-basis)**.** Given a level-$k-1$ CAD-cell $S \subset \mathbb{R}^{k-1}$, a point $(\alpha_1, \ldots, \alpha_{k-1}) \in S$ within that cell, and the bucket P_k of level-k polynomials from the projection (Def. 2.14), we call the set of single-variable polynomials

$$\{p(\alpha_1, \ldots, \alpha_{k-1}, x_k) \mid p(x_1, \ldots, x_k) \in P_k\}$$

the *lifting-basis* for cell S.

A lifting-basis as in Def. 2.23 is a preparation to isolate and compare roots—that's what lifting does. To construct it we select a level-$k-1$ cell and an arbitrary $k-1$ dimensional point within it. Then we take all the projection factor polynomials from the next level k and plug that point into every of those polynomials. The result is a set single-variable polynomials that only mention the last variable x_k.

For example, given the level-1 cell $(0, 1)$, a point $\alpha = 0.5 \in (0, 1)$ within it, and the level-2 polynomials

$$P_2 = \{x^2 + y^2 - 2, \quad xy - 1\},$$

we get the following lifting basis:

$$\{(0.5)^2 + y^2 - 2, \quad 0.5 \cdot y - 1\}$$

Analogous to difference between a projection-step and the projection as the final result, we define a single-lifting step in a notation more formal than McCallum [McC84, p. 43].

Definition 2.24 (Lifting-step)**.** Suppose we have a level-$k-1$ CAD-cell $S \subseteq \mathbb{R}^{k-1}$ as a sequence

$$[Comp_1, Comp_2, \ldots, Comp_{k-1}]$$

of section or sector cell components and point $(\alpha_1, \ldots, \alpha_{k-1}) \in S$ within that cell. Let that cell S be order-invariant on level-$k-1$ projection factor bucket P_{k-1} and let the projection factor bucket of level-k be

$$P_k = \{p_1, \ldots, p_l\},$$

containing l many level-k polynomials.

A lifting-step works as follows:

- Compute lifting-basis of S with respect to P_k.

$$\{p_1(\alpha_1^{k-1}, x_k), \ldots, p_l(\alpha_1^{k-1}, x_k)\}.$$

- Isolate all the real roots of each polynomial in the lifting basis and represent them as (single-variable) root-expressions. Let

$$\text{root}(p_{i_1}(\alpha_1^{k-1}, x_k), idx_{i_1}) \ < \ \ldots < \ \text{root}(p_{i_r}(\alpha_1^{k-1}, x_k), idx_{i_r}),$$

be those roots, ordered from smallest to highest, where $i_1, i_2, \ldots, i_r \in \{1, \ldots, l\}$ identifies the polynomial a root belongs to.

Then the cell components for level-k are

$$\begin{aligned}
Components_k := \{ & \\
-\infty < \quad & x_k \quad < \text{root}(p_{i_1}(x_1^k), idx_{i_1}), && \text{(sector)} \\
& x_k \quad = \text{root}(p_{i_1}(x_1^k), idx_{i_1}), && \text{(section)} \\
\text{root}(p_{i_1}(x_1^k), idx_{i_1}) < \quad & x_k \quad < \text{root}(p_{i_2}(x_1^k), idx_{i_2}), && \text{(sector)} \\
& x_k \quad = \text{root}(p_{i_2}(x_1^k), idx_{i_2}), && \text{(section)} \\
& \ldots & & \\
& x_k \quad = \text{root}(p_{i_r}(x_1^k), idx_{i_r}), && \text{(section)} \\
\text{root}(p_{i_r}(x_1^k), idx_{i_r}) < \quad & x_k \quad < +\infty, && \text{(sector)} \\
& \}, &&
\end{aligned}$$

and the lifted, level-k CAD cells—the lifting-step result—are

$$\{\ [Comp_1, Comp_2, \ldots, Comp_{k-1}, Comp_k] \mid Comp_k \in Components_k\}.$$

The lifting-step as in Def. 2.24 is formally complicated but conceptually simple. We select point within a level-$k-1$ cell and plug it into the polynomials of the level-k projection factor bucket P_k (see Def. 2.15), our lifting-basis. Then we isolate the real roots of those resulting single-variable polynomials and sort them from lowest to highest. The result of root isolation are algebraic reals represented by singl-variable root-expressions of the form

$$\text{root}(p_{i_1}(\alpha_1^{k-1}, x_k), idx_{i_1}).$$

The reason, why we use these indexes i_1, is to keep track of the multi-variable polynomial a root belongs to. We do this to restore the original polynomial in the root-expression after sorting, like

$$\text{root}(p_{i_1}(x_1^k), idx_{i_1}),$$

where the point α is not plugged in. Each of those multi-variable root-expressions becomes a section and two consective ones become bounds of a sector. So, the result of lifting a single level-$k-1$ cell are several level-k cells. Notice that the root-isolation of the lifting basis is only needed to compute the consecutive order among the polynomials in P_k with all their root indexes.

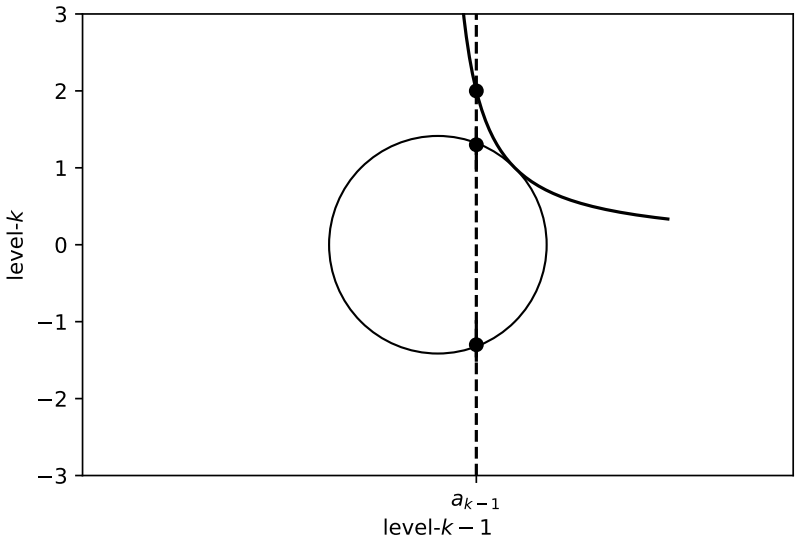


Figure 2.20: Qualitative root isolation. We plug a level-$k-1$ point into all level-k polynomials and isolate and compare their roots at that point—along the vertical line through α_{k-1}.

2.4.8 The full lifting

The full lifting, that creates a full decompostion of $\mathbb{R}^n$, is then obtained by applying the lifting step on each level from the bottom to the top.

Definition 2.25 (Lifting)**.** Given a projection factor set separated into buckets P_1 to P_n of same-level irreducible polynomials, construct the following sequence of CAD-decompositions:

1. D_1—ord-inv on $\mathbb{R}^1$ by base-case-construction using P_1
2. D_2—ord-inv on $\mathbb{R}^2$ by lifting each cell in D_1 using P_2
3. ...
4. D_{n-1}—ord-inv on $\mathbb{R}^{n-1}$ by lifting each cell in D_{n-2} using P_{n-1}
5. D_n—sign-inv on $\mathbb{R}^n$ by lifting each cell in D_{n-1} using P_n

Then D_n is the final full decomposition of $\mathbb{R}^n$ that is sign-invariant with respect to P_1 to P_n. Furthermore, the decompositions D_1 to D_{n-1} contain the "intermediate cells".

The notation in Def. 2.25 is adapted from Brown's improvement paper [Bro01, p. 451]. In using the projection by Collins [Col75] it suffices to repeat the lifting-step as in Def. 2.24 for each bucket P_2 to P_n in that order, because all the intermediate cells only need to be sign-invariant.

2.4.9 Order-invariance for McCallum-Brown CAD

However, when using the smaller and faster projection by McCallum [McC84] and its improvement by Brown [Bro01], the intermediate cells need to be what is called "order-invariant". This is a stronger property in the sense that if a CAD-cell is order-invariant, it will automatically be sign-invariant. Before defining "order" (see Def. 2.26) it is useful to make the following observations when using the McCallum-Brown-projection as in Def. 2.14:

- If a cell $S \in D_{k-1}$ is ord-inv on P_{k-1}, then applying the lifting step guarantees that D_k is sign-inv but not necessarily ord-inv on P_k.
- If a cell $S \in D_{k-1}$ is only sign-inv on P_{k-1}, then applying the lifting step does not guarantee that D_k is ord-inv or sign-inv on P_k.

The problem with the "normal" lifting-step is as follows: A sign-invariant decomposition D_1 of P_1 is order-invariant on P_1 [McC84, Assertion 1, p. 50]. But applying a "normal" lifting-step only guarantees that D_2 is sign-invariant on P_2 but not necessarily order-invariant—the first observation. So—by the second observation—the next lifting-step is not applicable, because it would produce a decomposition D_3 that is neither order-invariant nor sign-invariant, a useless decomposition. "At lower levels we need order-invariance to ensure that subsequent lifting steps are valid, but at the highest level there are no subsequent lifting steps, and thus sign-invariance is sufficient." [Bro05, p. 9].

As we have written, for the lifting to work correctly and to produce final CAD cells that are are guaranteed to be sign-invariant, the intermediate cells during the lifting phase must be "order-invariant", which is a stronger property than being sign-invariant.

Definition 2.26 (Order of a polynomial at a point)**.** Given a real polynomial $p \in \mathbb{Z}[\boldsymbol{x}_1^n]$ and a point $\boldsymbol{\alpha}_1^n \in S$ from an open space $S \subset \mathbb{R}^n$, and let k be the smallest, non-negative ($k \geq 0$) integer such that some k-th derivative g of p does not become zero at $\boldsymbol{\alpha}_1^n$, that is,

$$g(\boldsymbol{\alpha}_1^n) \neq 0.$$

Then we say that "polynomial p has order k at point $\boldsymbol{\alpha}_1^n$"

In earlier times the term "order" was used to refer to the degree of a polynomial (Def. 2.8). Today it is used to count the number of derivative operations: when we say that "g is a (partial) derivative of p of order k" or more concise "g is a k-order partial", we mean that g was obtained from p by successively applying the (partial) derivative operation k times. For example, the polynomial

$$p(x, y) := x^2 + y^2 - 2, \tag{11}$$

has the following partial derivatives of order $k = 0, 1, 2$:

$$
\begin{array}{lllll}
p^{(0)}: & p(x,y) = x^2 + y^2 - 2 & & & \\
p^{(1)}: & \dfrac{\partial p}{\partial x}(x,y) = 2x & & \dfrac{\partial p}{\partial y}(x,y) = 2y & \\
p^{(2)}: & \dfrac{\partial p}{\partial xx}(x,y) = 2 & & \dfrac{\partial p}{\partial yy}(x,y) = 2 & \dfrac{\partial p}{\partial xy/yx}(x,y) = 0
\end{array}
$$

Here, $p^{(k)}$ is called the set of all k-order partials.

Similarly, the definition of "order at a point" in Def. 2.26, adapted from McCallum [McC84] also counts a number of (partial) derivative operations. However, it refers to the smallest integer at which the polynomial's derivatives do not become zero, when we plug in the point in question. In other words, it counts the smallest order of derivatives at which a point is not a root anymore.

For example, the polynomial p of Eq. 11 has order 0 on point $(0, 0)$, because already on order 0, that is $p^{(0)}$, we have

$$p(0,0) = 1^2 + 1^2 - 2 = -2 \neq 0.$$

Note that the 0-th (partial) derivative of p is p itself. So 0 is the smallest order of derivatives at which the point in question is not a root.

Similarly, p has order 1 at point $(1, 1)$: On order 0 we have

$$p(1,1) = 1^2 + 1^2 - 2 = 0,$$

that is, it is a root of all polynomials of order 0. But because of

$$\frac{\partial p}{\partial x}(1,1) = 2 \cdot 1 = 2 \neq 0$$

there is a derivative of order 1 (in $p^{(1)}$) at which the point in question is not a root anymore.

As an important summary:

- A polynomial has order 0 at each point which is not a root and it has order of at least 1 at each root.

2.4.10 Nullifying cells' interference with order-invariance

The problem is that D_1 can contain a so-called "nullifying cell"; a term introduced by Brown and Košta [BK15].

Definition 2.27 (Nullifying cell)**.** A level-$k-1$ CAD-cell $S \subseteq \mathbb{R}^{k-1}$ is called a *nullifying cell* if there is a level-k polynomial $p(x_1, \ldots, x_k) \in P_k$ such that

$$p(\alpha_1, \ldots, \alpha_{k-1}, x_k) = 0 \text{ for all } \alpha_1^{k-1} \in S$$

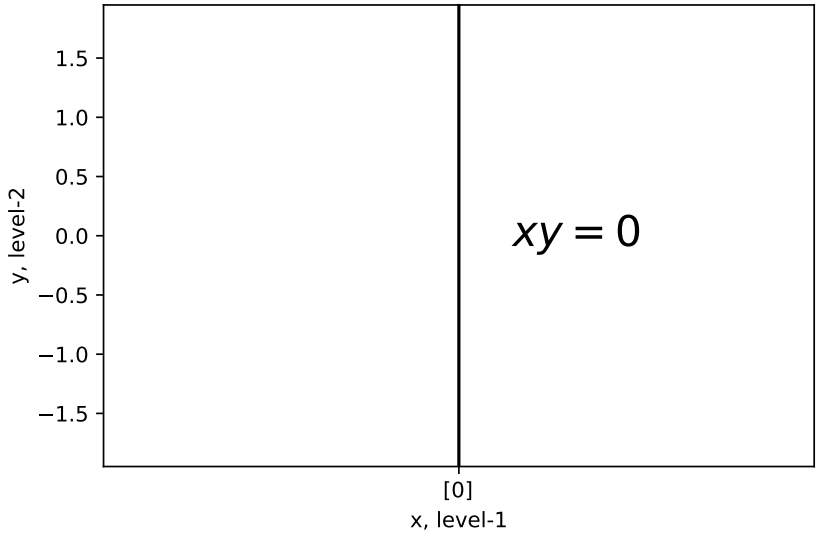


Figure 2.21: Qualitative root plot of level-2 polynomial $xy \stackrel{!}{=} 0$, which is nullified over level-1 cell [0].

A nullifying cell of level-$k-1$ is one where every of its points makes the level-k polynomial zero, even though we haven't plugged in a value for its last variable x_k. For example, plugging in the only point 0 from the level-1 cell $[0,0]$ into the level-2 polynomial $p(x,y) := xy$ yields $p(0,y) = 0$, that is, the resulting polynomial is zero along the whole y-axis as shown in Fig. 2.21.

The problem with a level-$k-1$ cell which nullifies a level-k polynomial is that the polynomial is sign-invariant over this cell—it is nullfied and therefore zero along the k-th axis—but is not necessarily order-invariant, which we need for subsequent lifting-steps [Bro05, p. 2].

As we will see, we can restore order-invariance in many situations by adding so-called "delineating polynomials" to the next-level bucket P_k. Whether this is possible will greatly depend on the "dimension of the cell".

Definition 2.28 (Dimension of CAD-cell)**.** Given a CAD-cell $S \subseteq R^n$ given as the sequence

$$[Comp_1, Comp_2, \ldots, Comp_n]$$

of sections or sectors. Then the number of sectors in that sequence is called the *dimension of the cell*.

There is a difference between the dimension of a cell and the dimension of the universe it lies in. In the extreme we have a single-point cell of dimension 0 that lies in the universe $\mathbb{R}^n$ of dimension n. We define the dimension of a cell as the number of sectors it contains, as in Def. 2.28. Intuitively, every section is a closed and every sector is an open inteveral along an axis so that only a

P_n:	$(i, \text{sign})^*$	$(i, \text{sign})^*$	...		
P_{n-1}:	(i, ord)	$(c, \text{sign})^*$	(r, ord)	...	
...					
P_2:	(r, ord)	$(c(c), \text{sign})^*$	(d, ord)	...	
P_1:	(i, ord)	(d, ord)	$(c(c), \text{sign})^*$	(r, ord)	...

Figure 2.22: Qualitative projection factors together with their invariance requirement: i = irreducible factor of an input polynomial, c = coefficient of a polynomial at some higher level, c(c) = coefficient of a derivative chain of only coefficients of other higher level projection factors, c(d) = coefficient of higher level discriminant or resultant polynomial, d = discriminant of a polynomial at some higher level, r = resultant of two polynomials of the same higher level, sign = sign-invariance, ord = order-invariance

sector "adds" to the dimension count. This definition works, because "cells are analytic submanifolds and thus homeomorphic to an open ball of the appropriate dimension" [BK15, p. 16].

Having the notion of a nullifying cell allows us to restate the difficulties with McCallum's lifting algorithm and the relationship between order-invariance and nullifying cells with more clarity. Recall that by Def. 2.25 we want to lift a an intermediate level-$k-1$ cell S_{k-1}, that is order-invariant to P_{k-1} by construction, into several level-k cells—let's call one such cell S_k—by using the next-level polynomials P_k. Our requirement is that these cells need to be order-invariant to P_k for subsequent lifting-steps, unless we lift into the final level ($k = n$, then sign-invarance is enough). So a lifted cell S_k, needs to be order-invariant to each $p \in P_k$. However, here we have an exception. As Brown points out [Bro01, p. 452] [Bro05, p. 10]: For the subsequent liftings-steps to be valid,

- order-invariance is only needed for p, if p has "a derivation as a resultant or discriminant of other projection factors".

This means that sign-invariance is enough for p if

- p belongs to the highest level of projection factors ($k = n$) or
- p has only a derivation of only coefficients of other projection factors.

These points are with * in Fig. 2.22, where we see full—but only qualitative—projection factor set separated into all levels. The level-n cells need only to be sign-invariant to the projection factors of P_n, because these are all input polynomials— polynomials from projecting these polynomials all have a smaller level. On the lower levels there are discriminants, resultants and coefficients of other discriminants or resultants. These polynomials require order-invariance of the intermediate cells at the corresponding level. Most notably there are also some (leading) coefficients that are derived as a chain of only (leading) coefficients from the input polynomials. Intermediate cells need only to be sign-invariant to these polynomials.

So, if p_k needs only sign-invariance on S_k, then lifting a nullifying cell S_{k-1}—which are by definition of the next-lower level—into a level-k cell S_k is no problem. Then

$$p(\alpha_1, \ldots, \alpha_{k-1}, x_k) = 0 \text{ for all } (\alpha_1^{k-1}) \in S_{k-1},$$

which is then also true for all points in the cylinder over S_{k-1}

$$p(\alpha_1, \ldots, \alpha_{k-1}, r) = 0 \text{ for all } (\alpha_1^{k-1}, r) \in S_{k-1} \times \mathbb{R}.$$

So that no matter into what cells the cylinder is sliced into, every cell that decomposition will produce a zero-sign in that polynomial p and therefore be sign-invariant.

2.4.11 Delineating polynomials to ensure order-invariance

The situation changes when the same nullifying cell S_{k-1} has to be lifted into an order-invariant cell, because while producing zero in a polynomial makes a cell sign-invariant, but not necessarily order-invariant. We summarize the following remarks by McCallum and Brown:

- If the nullifying cell has dimension 0 (see Def. 2.28), that is, if it is a single point, then we replace p by a "delineating polynomial" [Bro05, p. 3] in the projection factor bucket P_k to restore and guarantee order-invariance;
- If the nullifying cell has a positive dimension, we check if every point in the cylinder over that cell is already order-invariant on p. If not, then the lifting fails and abort the McCallum-Brown CAD construction [Bro05, p. 6].
- If a cell is not nullifying for any polynomial of the next-higher level, then lifting automatically produces order-invariant cells of the next level. (See McCallum's "Lifting Theorem" [McC84, Thm. 3.2.1, p. 45] [McC84, Thm. 3.2.3, p. 47]).

We will discuss the first two situations in more detail. First, if the nullifying cell has dimension 0 and order-invariance for the lifted cell, then we can correct the situation and guarantee order-invariance by introducing a so-called "delineating polynomial" [Bro05, p. 3].

Definition 2.29 (Delineating polynomial)**.** Given a nullifying, level-$k-1$, single-point cell $\alpha_1^{k-1} \in \mathbb{R}^{k-1}$ (of dimension 0) and a nullified, level-k polynomial

$$p(x_1, \ldots, x_k) \in P_k \text{ with } p(\alpha_1, \ldots, \alpha_{k-1}, x_k) = 0,$$

which has order t on α, that is, t is the smallest order of derivatives at which α is not a root anymore. Let $p^{(t)}$ be set of all t-order partials and

$$p_\alpha^{(t)} \;=\; \{\; q(\alpha_1, \ldots, \alpha_{k-1}, x_k) \mid q(x_1, \ldots, x_k) \in p^{(t)} \;\}$$

be these partials at point α. Then

$$d_p(x_k) \;:=\; \gcd(p_\alpha^{(t)})$$

is called a *delineating polynomial*. Furthermore d_p is a level-k polynomial that only mentions the variable x_k.

The purpose of the level-k, single-variable delineating polynomial d_p in Def. 2.29 is to replace the nullified polynomial $p \in P_k$ of the same level and to ensure that the cylinder $(\alpha \times \mathbb{R})$ over the nullifying cell of dimension 0, that is, the line along the k-axis in Fig. 2.21, is cut at the points, where the order of p at α is not t.

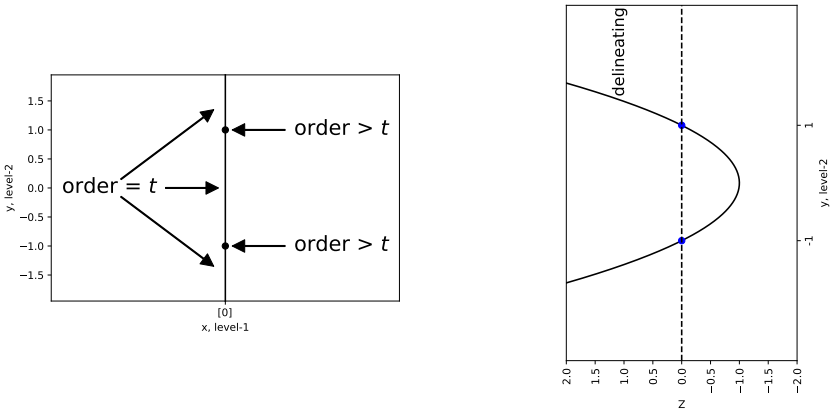


Figure 2.23: Qualitive root plot (left) of a level-2 polynomial (vertical line) that is nullified by a level-1 cell [0], and a qualitative, regular graph of a single-variable, delineating polynomial (right). The polynomial one the left has order t for all points along the vertical line except for the two marked ones. Notice that the level-2 coordinates of the roots on the right are the same level-2 coordinates of the points on the left where the order is greater than t.

This is shown in Fig. 2.23. We know that p has order t at almost all points of that cylinder, but at finitely points it may have a higher order [Bro05, p. 3]. Each of such points is a common root of the t-order partials $p_\alpha^{(t)}$—otherwise such a point would have order t—and because $d_p(x_k)$ is the greatest common divisor polynomial of these partial derivatives has precisely those common roots as its roots. And because d_p replaces p in P_k and the lifted level-k cells are split according to the roots of polynomials in P_k, they will be split at those points, where the order is higher. Thus, the order will be the same in each lifted cell, and order-invariance of these level-k cells to P_k can be guaranteed.

Second, if the nullifying cell has a positive dimension and there is a possibility that the lifted cells may not be order-invariant, then there is nothing we can do but to check more rigorously and hope that the whole cylinder over the nullifying cell is already order-invariant. If that's the case, any decomposition of that cylinder and thus any the resulting lifted cells will also be order-invariant. This test however is costly and would require another computation intensive invocation of a CAD algorithm [Bro05, p. 6]. Brown suggests to use a faster, but

incomplete test algorithm that may sometimes fail to prove that the cylinder is order-invariant even though it is [Bro05, p. 7]. For our purposes, we simply abort the CAD construction when we encounter nullifying cell of positive dimension.

As an important summary:

- After using McCallum projection, or McCallum-Brown projection, the lifting can fail when there is a nullifying cell of positive dimension.

2.4.12 When order-invariance can always be ensured

McCallum noticed that there are no nullifying cells of positive dimension when the input polynomials are "well-oriented" [McC84, p. 92].

Definition 2.30 (Well-oriented polynomial)**.** A polynomial $p \in \mathbb{Z}[\boldsymbol{x}_1^n]$ with

$$p(x_1^{n-1}, x_n) \;:=\; p_k(x_1^{n-1}) \cdot x_n^k + \ldots + p_1(x_1^{n-1}) \cdot x_n^1 + p_0(x_1^{n-1}) \cdot x_n^0,$$

where the solution space of the system of zero-equations with the coefficients

$$p_k(x_1^{n-1}) = \ldots = p_1(x_1^{n-1}) = p_0(x_1^{n-1}) = 0$$

has dimension 0, is called *well-oriented*.

So for the subset of well-oriented polynomials his CAD algorithm is complete and does not fail. The formulation of the well-oriented-definition in Def. 2.30 is adapted from Brown [Bro05, p. 453]. McCallum noted that all polynomials up to level 3, that is, with three or less variables, are automatically well-oriented [McC84, p. 92].

This concludes our description of a full Cylindrical Algebraic Decomposition.

2.5 Single Cylindric Algebraic Cells

The SMT solving framework NLSat—for the "Non-linear theory over the reals"—by Jovanović and de Moura [JM13] was created to check if a Boolean combination of polynomial (in-)equalities is satisfiable. It inspired a special application of CAD, namely, to construct a single cell around a point. Roughly, the main idea is to guess a potentially satisfying point and if it turns out to be an infeasible point—one that does not satisfy all polynomial constraints—, to generalize that to a region of infeasible points, so that one may not guess any point within that region again.

While Jovanović and de Moura provided the first single-cell-construction-algorithm, the problem and the benefits of a single cell construction have been further and more intensively studied by Brown. One of his results was a more efficient algorithm to construct "open CAD-cells" [Bro13], which have the same dimension (see Def. 2.28) as the universe they lie in. He later, generalized his improved construction with Košta, to arbitrary, regular CAD-cells [BK15] and called this algorithm "OneCell". In the following we discuss this "OneCell" construction.

2.5.1 Construct a single cell around a single point

Roughly, the main task of OneCell is the following: Given a (finite) set of real polynomials $P \subset \mathbb{Z}[\boldsymbol{x}_1^n]$ of mixed levels and a point $\alpha \in \mathbb{R}^n$, construct of a single CAD-cell S, that is sign-invariant on P, around α, that is, the point $\alpha \in S$ should be included in S. Furthermore, because returning the single-point cell $S = \{\alpha\}$ would be a trivial solution to the stated problem—because such a cell is a valid CAD-cell—, there is another requirement that this cell should be as big as possible, namely, at least as big as the corresponding cell around the same point in a full McCallum-CAD-decomposition [BK15, Item 2,p. 16].

Even though we can use CAD without SMT to find solutions for the input constraints, mixing SMT with CAD has one main advantage:

- Normal SMT solvers are highly optimized to find and use only the relevant input constraints through reasoning in the Boolean abstraction. They would use CAD mostly on a subset of the input constraints to check if a subset is satisfiable. In contrast, NLSat uses CAD to explain and generalize an unsatisfiable subset of constraints. Nevertheless, using CAD with as few constraints as possible can save a lot of computation time, because fewer constraints mean fewer polynomials and possibly fewer polynomial variables to process. This is important, because a full CAD's computation time grows doubly-exponential—and therefore scales badly—in the number of variables [BD07].

So we can replace computing one large CAD instance with computing several but much smaller instances.

And even though we can use SMT with a full decomposition CAD algorithm that simply selects the single cell around the point in question, mixing SMT with a specialized single-cell-construction has two main advantages:

- Compared to a full composition, constructing a single cell requires less projection factors to be computed in the projection-phase—as we will explain further in this section—and therefore much less computation time. And it only ever needs to lift a single cell in the lifting-phase. Actually, instead of extending a lower-level cell into a higher-level cell through lifting, we will use a highest-level, all-encompassing cell and make it appropriately smaller and smaller.

- Compared to a full decomposition, constructing a single cell can result in a larger cell—as we will see further in this section—and a"larger cell means a stronger generalization" of an infeasible point [Bro13, p. 1].

2.5.2 The OneCell algorithm

We implemented the OneCell algorithm as presented by Brown and Košta [BK15]. Because the pseudocode for this algorithm is several pages long, we refrain from restating it here in full detail and refer to their publication. In brief, OneCell works as follows: We start with a large, all-encompassing, single CAD-cell that covers the whole real space $\mathbb{R}^n$. We then process the input polynomials one by one and "merge" them into cell, while we ensure that our single cell stays sign-invariant to the polynomials processed so far and contains the point α. Notice that the initial $\mathbb{R}^n$-covering cell ensures this: It can be

seen as sign-invariant to the empty poly-set and it contains—since it contains all points—the point α. To "merge" a polynomial we check whether its roots cross through our cell. If they do, we shrink our cell appropriately along several dimensions so that these roots become the new cell's bounds—recall that the roots of polynomials make up the bounds of a CAD-cell—and sign-invariance is ensured. Brown and Košta call this a "refinement". The result is a single CAD-cell around point α that is sign-invariant to all input polynomials.

The advantages of this OneCell algorithm can be conveyed both algebraically and geomtrically. First, we present the advantages algebraically: OneCell uses a McCallum-Brown projection operator as for a normal CAD (see Def. 5.2), which computes leading coefficients, discriminants and resultant, and builds up a projection factor set separated into level buckets (see Def. 2.15. However, because this algorithm processes and projects the input polynomials not all at once but one after the other, we can avoid a lot of resultant computations—this is the major reason in the computation time reduction. Recall that in a normal CAD projection-step we take all polynomials of a certain level and compute—among the coefficients and discriminants of each polynomial—the resultant between any two pairs of those polynomials. This means that if we have k polynomials in that level, we will compute $\mathcal{O}(k^2)$ many resultants for that level. In later projection-steps we compute the pairwise resultants of those resultant and so forth, which is particularly costly.

P_n:	p_1^*			
$\ldots$:				
P_k:	q_1^*	q_2^*	i	
P_2:	r_1	r_2^*	$\mathrm{res}(q_1, i)^{1.,*}$	$\mathrm{res}(q_2, i)^{3.}$
P_1:	s_1^*		$\mathrm{res}(r_2, \mathrm{res}(q_1, i))^{2.}$	$\ldots^{4.}$

Figure 2.24: Qualitative resultant computation order of the projection factor set for a new input polynomial i of level k. We compute the resultant of i —and its descendents—against the marked polynomials on the same level in a depth-first manner. At each level there are at most two marked polynomials—representing the cell bounds at that level—to compute the resultant against—except for P_1, where no further resultant computation is needed.

In the OneCell context, we also compute a projection factor set level by level, which we visualize in Fig. 2.24. However, we don't start at the highest level, but at the level k of the next input polynomial—named i in the figure. At each level, we have at most two polynomials that are part of the cell's bounds—marked with * in the figure—,assuming that we already processed some input polynomials like $p_1, q_1, q_2, r_1, r_2, s_1$. We then compute the resultant of the next input polynomial i only against the marked polynomials of the same level in a depth-first search manner.

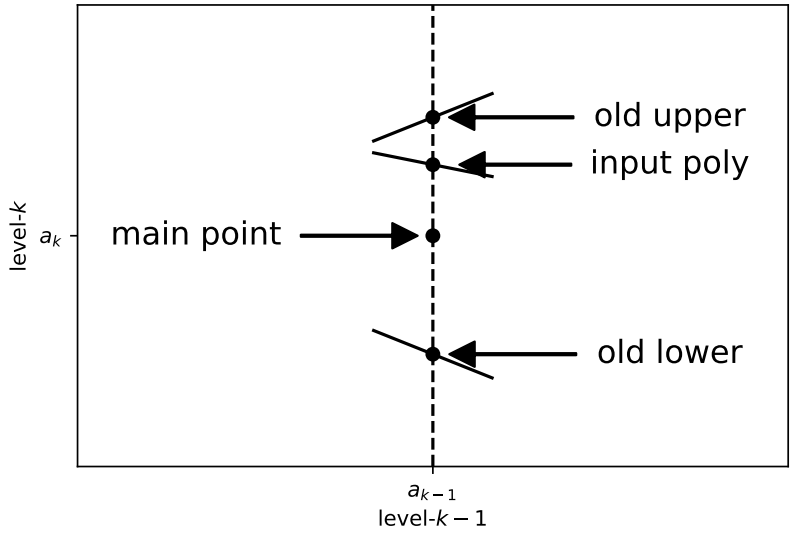


Figure 2.25: Qualitative root isolation when we process a a new input polynomial of level k, but already have existing level-k cell bounds. This polynomial becomes a new bound if one of its roots, isolated at level-k (along the vertical line), lies closer to α_k than the existing bounds.

In more detail: We assume that we already have processed some input polynomials, created an intermediate projection factor set and have marked those polynomials at each level that are involved in the representation of our single cell. When we process the next input polynomial i of level k, we add it to our projection factor set at level k. We compute its leading coefficient, its discriminant—we ignore those two for in Fig. 2.24 for simplicity—and the resultant against the marked polynomials q_1 and q_2, but we don't add them to projection factor set just yet. We first check if we have to "refine" the cell at level k, that is, we check if the current input polynomial becomes one of the cell's bounds. Let's assume that cell's component at level k is a sector so that the two marked polynomials are used to represent the sector's lower and upper bounds as

$$\text{root}(\ q_1(x_1,\ldots,x_{k-1},z), idx_{q_1})\ ) \quad < x_k < \quad root(\ q_2(x_1,\ldots,x_{k-1},z), idx_{q_2}\ ).$$

Recall that this is our way (see Multi-variable root-expression in Def. 2.21) to represent a cell's bounds for the k-th variable. These bounds are more intuitively written in the form

$$f^{low}(x_1,\ldots,x_{k-1}) \quad < x_k < \quad f^{high}(x_1,\ldots,x_{k-1}).$$

We do this refinement check by isolating the roots of the bound polynomials and input polynomials along the x_k-axis. In Fig. 2.25 we see how this works: We plug the first $k-1$ components of our point α into the polynomials, isolate their roots—visually where they cross the vertical line over α_{k-1}—and check

if any of the input-polynomial's roots lies closer to the k-th component of α than the cell's existing bounds. These existing bounds are isolated roots of the marked polynomials and correspond to the evaluated root-expressions

$$\underbrace{\text{root}(\ q_1(\alpha_1,\ldots,\alpha_{k-1},z),idx_{q_1})}_{\in\mathbb{R}}\ ) \quad < \underbrace{\alpha_k}_{\in\mathbb{R}} < \quad \underbrace{\text{root}(\ q_2(\alpha_1,\ldots,\alpha_{k-1},z),idx_{q_2}\ )}_{\in\mathbb{R}}.$$

If a root of the input polyomial i indeed lies closer, this i becomes a new bound. In Fig. 2.25 one root of the input polynomial i lies closer to α_k than the upper bound q_2, so this i replaces the q_2 as the new upper bound; and the cell becomes smaller. This is called a "refinement".

We finish our explanation of Fig. 2.24: After this refinement, the resultants we computed—and the leading coefficient and the discriminant for that matter—is processed as if it was an input polynomial. But since the resultant has always a lower level, we repeat the process at a lower level. This why we can view this form of processing input polynomials as a depth-first search variant to construct the projection factor set.

In total we will compute at most two resultants for each input polynomial at a certain level. So, if we have k polynomials in that level, we only compute $\mathcal{O}(2 \cdot k)$ many resultants.

- Computing the resultant of a new input polynomial against the (at most two) cell bound polynomials of the same level, is sufficient only because we construct a single cell.

For a full CAD we would still have to compute the resultant between any two pairs of polynomials at the same level, even if we processed the input polynomials one by one.

To understand why at most two polynomials at each level are necessary to represent a cell's bounds, recall that a cell is a sequence of cell components—each a section or a sector (see Def. 2.20). The bounds of a level-k component are always represented by polynomials of the projection factor set at level k. So these bounds involve the following number of polynomials in the following cases:

- 0 polynomials if the component is a sector

  $$-\infty \quad < x_k < \quad \infty$$

  with only infinity bounds.

- 1 polynomial if the component is either
  - a sector

    $$\begin{aligned} -\infty \quad &< x_k < \quad \text{root}(p(\boldsymbol{x}_1^k),\text{idx}),\ \text{or} \\ \text{root}(p(\boldsymbol{x}_1^k),\text{idx}), \quad &< x_k < \quad +\infty \end{aligned}$$

    with one polynomial bound and one infinity bound, or
  - a section

    $$x_k \ = \quad \text{root}(p(\boldsymbol{x}_1^k),\text{idx}),$$

    which always has only one polynomial bound, or

– a sector

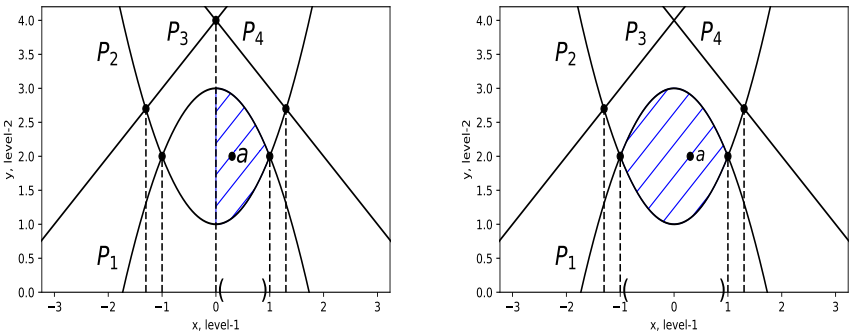
$$\text{root}(p(\boldsymbol{x}_1^k), \text{idx}_1) \quad < \; x_k \; < \quad \text{root}(p(\boldsymbol{x}_1^k), \text{idx}_2),$$

with the same polynomial in both bounds but with different root indices.

- 2 polynomials if the component is a sector

$$\text{root}(p(\boldsymbol{x}_1^k), \text{idx}) \quad < \; x_k \; < \quad \text{root}(q(\boldsymbol{x}_1^k), \text{idx}),$$

with two polynomial bounds.



(a) We see the small, constructed cell around point α when the resultants are computed between all pairs of polynomials p_1 to p_4 (marked points). Here, $(0, 1)$ is the level-1 bound.

(b) The construced cell around α when the polynomials p_1 to p_2 are processed in that order. The cell is now larger, because $\text{res}(p_3, p_4)$ is not computed. Here, $(-1, 1)$ is the level-1 bound.

Figure 2.26: Qualtitive root plot of 4 polynomials p_1 to p_4. We compare the cell size when computing all pairwise resultants (regular CAD) versus computing resultants against cell bound polynomials (OneCell). In both cases P_1 and P_2 are the cell's level-2 bounds.

As we mentioned before, the advantages of the subsequent processing of the input polynomials, and the fact that we compute the resultants only against the two marked polynomials that represent our single cell bounds, can also be shown geometrically. This is visualized in Fig. 2.26, where we assume to have polynomials named p_1 to p_4 of level-2. On the left we compute the resultants like in a full CAD algorithm (see Sec. 2.2). Recall that resultant between two polynomials of level-k is a polynomial of level-$k-1$ whose roots indicate where the two intersect (see Def. 2.10) and forces us to create more but smaller intermediate level-$k-1$ cells to ensure a valid full CAD. We can see that the polynomials p_1 and p_2 represent the bounds at level-2, that is, along the y-axis. We can also see that the resultant between p_3 and p_4 makes up the lower bound to the left of the cell at level-1, that is, along the x-axis, and that the resultant between p_1 and p_2 makes up the upper bound to the right. In the case of a full CAD the resultant between p_3 and p_4 is necessary to ensure that all cells in the cylinder over the level-1 cell are sign-invariant. However, the main requirement

for a cell is that it should be sign-invariant to all input polynomials and because p_3 and p_4 do not cross the cell, the cell can actually be larger and still be sign-invariant. Actually it can be as large as on the right of Fig. 2.26. There we process p_1 first, which becomes the lower bound at level-2. Then we process p_2, which becomes the upper bound at level-2. We compute the resultant of those two, which shrinks the cell at level-1 to the highlighted intervall. Afterwards we process p_3 and compute its resultant against p_1 and p_2. However, the bounds inflicted by the resultant on level-1 lie outside of the cell. So they have no effect on our cell. And because p_3 lies outside of the cell on level-2 as well, it does not become part of the level-2 bounds. This is why when we process p_4, we only compute the resultants against the still active cell bounds p_1 and p_2, but not against p_3. And finally because p_4 and the bounds inflicted by its resultants lie outside of the cell on level-2 and level-1, they have no effect on our cell. So p_4 does not become part of the level-2 bounds as well. The main improvement of OneCell is the following:

- Some resultants may not be computed and the constructed cell can be larger compared to a full CAD.

3 Model Constructing Satisfiability Calculus

The Model Constructing Satisfiability Calculus (MCSAT) by Jovanović and de Moura [MJ13] is a relatively new approach to tackle Satisfiability Modulo Theories (SMT) problems in a more efficient manner than previous ones.

3.1 SMT formulas over Non-linear Real Arithmetic

Solving SMT problems is concerned with checking if a Boolean combination of constraints is satisfiable. When the constraints are polynomial (in)equalities, such a combination looks as follows:

$$\underbrace{x = 2}_{\text{constraint}} \quad \wedge \quad \underbrace{(\underbrace{x^2y + 2}_{\text{poly}} < \underbrace{xy}_{\text{poly}})}_{\text{constraint}} \quad \vee \quad \neg(\underbrace{y^2 \geq 3}_{\text{constraint}}). \tag{12}$$

SMT solving is related to mathematical logic and theorem proving, and therefore uses a lot of vocabulary from that field. It also helps us to describe the construct in Eq. 12 with more precision. In mathematical logic terms: In SMT solving we check if a first-order, quantifier-free formula over a specific theory has a model.

3.1.1 Non-linear polynomial formulas

The construct in Eq. 12 is a formula over the theory of **Non-linear Real Arithmetic** (NRA)—our theory of choice for this thesis. However, to simplify the presentation and processing of such a formula, we assume for the rest of this thesis that it is transformed into an equivalent formula that only contains polynomial (in)equalities against zero as in:

$$\underbrace{F(x,y)}_{\text{Name + free variables}} := \underbrace{x - 2 = 0}_{\text{atom}} \quad \wedge \quad \underbrace{(\underbrace{x^2y + 2 - xy}_{\text{term}} < \underbrace{0}_{\text{term}})}_{\text{atom}} \quad \vee \quad \neg(\underbrace{y^2 - 3 \geq 0}_{\text{atom}}). \tag{13}$$

Using vocabulary from mathematical logic, an SMT-formula like the one in Eq. 13 has the following characteristics:

- It is a "formula", that is, it is built using logical/Boolean connectives like $\wedge, \vee, \neg$ if it contains more than one atom. We represent a formula by a capital letter F or G.
- It is "quantifier-free", that is, it mentions no quantifier like $\exists$ or $\forall$; all its variables like x and y are unbounded and free. We call these "theory variables", because the values they accept depend on the underlying theory. We use names like x, y and z.
- It is first-order, which means:
  - Its smallest parts that can evaluate to truth values, called atomic formulas or "atoms" for short, can have structure and may not merely be Boolean variables.

  - Its free variables like x and y only accept values from the "domain of discourse" from the specified theory.

- It has a specific theory, which specifies the allowed logical Boolean connectives, the allowed relational symbols like $<$ or $=$ and their interpretations, the allowed functional symbols like $+$ or $\cdot$ and their interpretations, and the domain for the free variables x and y, usually called the "domain of discourse". It also specifies the syntactic structure of the terms in the atoms. These are all things that can normally be interpreted arbitrarily in regular first-order logic. In our SMT case we fix and use the theory of Non-linear Real Arithmetic (NRA), which means:

  - We can use any logical Boolean connectives, because there are no restrictions here.
  - The domain of discourse is the real numbers $\mathbb{R}$ and so the free variables in F only accept real numbers; So, they are "real variables".
  - The allowed relational symbols are $=$, $\neq$, $<$, $\leq$, $>$ and $\geq$, and their intepretation is fixed to usual interpretation with real numbers like $=_{\mathbb{R}}$ and $<_{\mathbb{R}}$. For example, $1 =_{\mathbb{R}} 1$ and $1 <_{\mathbb{R}} 2$ are true, and $1 =_{\mathbb{R}} 2$ and $2 <_{\mathbb{R}} 1$ are false as usual.
  - The allowed functional symbols are $+$, $-$, $\cdot$ and exponentiation $\cdot^n$ with a natural number, and their intepretation is fixed to usual interpretation with real numbers like $+_{\mathbb{R}}$ and $\cdot_{\mathbb{R}}$. For example, $1 +_{\mathbb{R}} 1$ is 2 and $1 \cdot_{\mathbb{R}} 1$ is 1 as usual. So, the theory prescribes the usual arithmetic.
  - The syntactic structure of the terms are fixed to polynomials with several variables. So the theory allows to build and use (the polynomial fragment of) "non-linear" functions—it could also be called "Polynomial Real Arithmetic".

- It has interpretations and can have a model. An interpretation contains a domain of discourse, an assignment of the free variables in F to values of that domain, and an assignment of the abstract relational and functional symbols to concrete relations and functions. One interpretation can be completely different from another in all those points.

  - If an interpretation makes F true, this interpretation is called a model. And F is called satisfiable.
  - If all possible interpretations make F false, F has no model and is called unsatisfiable.

  However in our case, because the theory of NRA fixes everything except the assignment of the free variables x and y, an interpretation—and hence also a model—of F reduces to a variable assignment of the free variables to real numbers. So, an interpretation of F is only different from another in the real numbers it assigns to x and y.

SMT solving in its essence is theory agnostic and can be used with almost any theory, even a mixture of several theories [BHM09, p. 849f]. In the abstract setting we call an atomic formula an "atom", or "constraint". But because in this

thesis we restrict ourselves to the theory of Non-linear Real Arithmetic (NRA) with the implications described previously, a particular computation time intesive problem, those constraints are polynomial equalities and inequalities, which we simply call "polynomial constraints". Thus more precisely, we work on what is also known as the Quantifier-Free Non-Linear Arithmetic fragment of first-order logic (QF_NRA).

3.1.2 Basic SMT vocabulary from mathematical logic

In its relation to theorem proving, SMT solving is equivalent to proving existentially-only-quantified formulas, called the existential fragment of NRA, because

$$F(x_1, \ldots, x_n) \text{ has a model} \quad \text{if and only if} \quad \exists x_1 \ldots \exists x_n \colon F(x_1, \ldots, x_n) \text{ is valid.}$$

In the SMT context we are guaranteed to get satisfying values for these variables x_1 to x_n if the formula has a model, whereas in the theorem proving context we may be able to prove that such satisfying values for the x's exist without knowing what values specifically. As an aside: In Philosophy this is known as "knowing that" without "knowing what".

For many theories there are "decision procedures", that is, algorithms that tell us, always in a finite amount of time, whether a problem has a solution or not, or—in mathematical logic terms—whether a formula is valid or not. For many theories there are even efficient—polynomial time—decision procedures, although sometimes only when we restrict ourselves to constraints connected by logical AND $\wedge$, thereby losing some of the expressiveness. For the theory of NRA there also exist decision procedures but to the best of our knowledge no efficient ones. Cylindrical Algebraic Decomposition (see Sec. 2.2) is part of many of such procedures. It was invented as part of a Quantifier-Elimination (QE) procedure [Col75] to remove quantifiers from formulas of the form

$$\exists x_1 \ldots \exists x_n \colon F(x_1, \ldots, x_n),$$

as described before, although it also works with $\forall$ quantifiers. As a QE method it can be used to eliminate all quantifiers and to check such a formula for validity. However, it is computationally quite slow. If we use it on formulas that are only constraints connected with the logical AND, it is more efficient but still quite slow. As described later in this section, instead of one large invocation of an CAD decision procedure on the whole formula, we use CAD in its "raw" form for multiple small invocations on parts of the formula, which in practice is a lot more efficient as Jovanović and de Moura have shown in [JM13].

The main focus SMT is to extend "ordinary" Satisfiability solving (SAT) in its expressiveness while reusing its highly-efficient solving techniques. Ordinary SAT solving only works on propositional, Boolean formulas. In propositional formulas the atoms are plain propositional, Boolean variables—to be assigned true and false—with no further inner structure. We obtain a propositional formula from an SMT-formula like

$$F(x, y) := \; (xy < 3) \quad \wedge \quad \neg(xy < 3) \quad \wedge \quad 3x^2 + 3y^3 = 4 \tag{14}$$

by replacing the same atoms by the same Boolean variable and different atoms by different Boolean variables:

$$G(b, c) := \; b \quad \wedge \quad \neg b \quad \wedge \quad c. \tag{15}$$

- This is called the Boolean abstraction of an SMT-formula.

Both have different variable assignments.

Definition 3.1 ((Propositional) Variable assignment)**.** Given a prositional formula $G(b_1, \ldots, b_n)$ with Boolean variables b_i, then a (Boolean) variable assignment is a function

$$\vartheta \colon \{b_1, \ldots, b_n\} \to \{\text{true}, \text{false}\},$$

where the evaluation of G under this assignment

$$\vartheta(G) = G(\vartheta(b_1), \ldots, \vartheta(b_n))$$

is the result of replacing the Boolean variables by the assigned truth values.

We can represent a Boolean variable assignment by a sequence of the variables or their negation. For example

$$[b_1, b_2, \neg b_3]$$

represents the variable assignment that maps b_1 and b_2 to true and b_3 to false.

Definition 3.2 ((Real) variable assignment)**.** Given a QF_NRA-formula $F(x_1, \ldots, x_n)$ with real variables x_i, then a variable assignment is a function

$$\vartheta \colon \{x_1, \ldots, x_n\} \to \mathbb{R},$$

where the evaluation of F under this assignment

$$\vartheta(F) = F(\vartheta(x_1), \ldots, \vartheta(x_n))$$

is the result of replacing the variables by the assigned real numbers.

We can represent a variable assignment by a sequence of real number mappings like

$$[x_1 \to 0, \quad x_2 \to 0, \quad x_3 \to 1.1].$$

Definition 3.3 (Partial assignment)**.** Given a QF_NRA-formula $F(x_1, \ldots, x_n)$ with real variables x_i, then a partial assignment is a variable assignment where some of the x_i are unassigned.

As we will see, MCSAT is going to create larger and larger partial assignments that, if the formula in question is satisfiable, will ultimately become a (full) satisfying assignment.

Coming back to the Boolean abstraction, we can say reasoning on the Boolean abstraction is often much faster than in the theory and sometimes sufficient to derive unsatisfiability of the SMT-formula. With "reasoning" we mean to derive entailed formulas—those that follow logically from the given formula—such as the constant formula false. With

$$F(x, y) \models \text{false}$$

state that the left part entails the right part, that is, for every variable assignment that satisfies the left part it also satisfies the right part. If we are able to derive false, we have proven that $F(x, y)$ is unsatisfiable, because false by definition has no satisfying assignment. For example, in the Boolean abstraction in Eq. 15 we cannot satisfy b and $\neg b$ at the same time. So, we derive false by its Boolean structure alone and so prove that the SMT-formula in Eq. 14 is unsatisfiable without every having to use real arithmetic.

The subtle difference between the entailment symbol $\models$ and the logical implication operator $\implies$ is that the first describes a semantic relationship—think of variable assignments—between two formulas, and the second is only a syntactic representation of this relationship inside a formula. We also use the $\models$ symbol to express that a formula is valid.

Definition 3.4 (Validity of a formula)**.** A QF_NRA-formula $F(x_1, \ldots, x_n)$ is valid and written

$$\models F(x_1, \ldots, x_n)$$

if the following holds:

$$\text{true} \models F(x_1, \ldots, x_n),$$

that is, if F evaluates to true under every variable assignment—because the formula true is by definition true under every assignment.

In other words, a valid formula like

$$(x > 0) \quad \vee \quad (x \leq 0)$$

is "universally true". The following insight will be used in MCSAT:

- If a formula $F(x_1, \ldots, x_n)$ is valid, then it logically follows from every other formula, that is, for every formula G we can write

$$G(x_1, \ldots, x_n) \models F(x_1, \ldots, x_n).$$

Furthermore, because of

$$F(x, y) \models \text{false} \text{ if and only if } \models (F(x, y) \implies \text{false})$$

we can rephrase a question of entailment into a question of validity. So, when you syntactically create and manipulate a formula, you use $\implies$. When you state some semantic property of a formula like validity, you use $\models$.

In mathematical logic there are widely-known and used so-called normal forms, which describe a certain syntactical shape which a formula can be transformed into and from which some properties are more easily visible. A formula is in Disjunctive Normal Form (DNF) if it has the syntactic shape of

$$(L \wedge L \wedge \ldots \wedge L) \quad \bigvee \quad (L \wedge L \wedge \ldots \wedge L) \quad \bigvee \quad \ldots \quad \bigvee \quad (L \wedge L \wedge \ldots \wedge L),$$

where the big logical connectives are disjunctions—logical OR operator $\vee$—and each L is a distinct literal and the number of literals in each "min-term" $(L \wedge L \wedge \ldots \wedge L)$ and the literals themselves inside it can be completely different from another min-term. If there are zero min-terms, the formula stands for false.

Definition 3.5 (Literal)**.** A *literal* is either an atomic formula A or the negation $\neg A$ of one.

In a propositional formula an atomic formula is simply a Boolean variable and this Boolean variable or its negation are both called a "literal". In the context of the theory of NRA an atom is a single polynomial equality or inequality so that the (in-)equality

$$(xy < 3) \text{ and its negation } \neg(xy < 3)$$

are both called "literals". If a formula in DNF it is rather easy to check if it is satisfiable. Go through each min-term and check if it is satisfiable: If we find a single satisfiable min-term, then the whole formula is satisfiable. And checking a min-term is much easier, because only conjunctions—logical AND operator $\wedge$—of literals are involved. However, converting an arbitrailly syntactically shaped formula into DNF is hard. It must be, otherwise SAT and SMT would obviously be much easier to solve.

It the context of SAT and SMT a formula to check is usually given and processed in a Conjunctive Normal Form—or it can be easily transformed into this form.

Definition 3.6 (Conjunctive Normal Form (CNF))**.** A formula is in *Disjunctive Normal Form* (DNF) or *clause-form* if it has the syntactic shape of

$$(L \vee L \vee \ldots \vee L) \bigwedge (L \vee L \vee \ldots \vee L) \bigwedge \ldots \bigwedge (L \vee L \vee \ldots \vee L),$$

where the big logical connectives are conjunctions—logical AND operator $\wedge$—and each L is a literal and the number of literals in each "clause" $(L \vee L \vee \ldots \vee L)$ and the literals themselves inside it can be completely different from another clause. If there are zero clauses, the formula stands for true.

Definition 3.7 (Clause)**.** A formula is a *clause* if it has the syntactic shape of

$$(L \vee L \vee \ldots \vee L),$$

where each L is a distinct literal, that is, an atom A or the negation $\neg A$ of one. The empty clause, containing zero literals, stands for false.

A clause is satisfied if one of the literals is satisfied by a variable assignment. We have to satisfy all the clauses at the same time in a CNF formula, which means that we have to find a variable assignment that satisfies at least one literal in every clause. If a formula is given in CNF, it is not easy to see and find such a satisfying assignment. However, it is easy to see assignments which *do not* satisfy this formual. Indeed

- for every single clause it is easy to see an unsatisfying, propositional partial assignment, unless it contains an atom A and its negation $\neg A$, because containing $A \vee \neg A$ makes the clause is universally true. Nevertheless, an unsatisfying, partial assignment forbids possibly more (full) assignments.

For example, the clause

$$\ldots \bigwedge (a \vee \neg b \vee c) \bigwedge \ldots$$

inside a larger formula—with possibly more Boolean variables than a, b and c—forbids the partial assignment $\{a \to \text{false}, b \to \text{true}, c \to false\}$. In other words, this clause forbids all (full) assignments that contain this specific partial assignment. Similary, in the context of the theory of NRA, a clause like

$$\ldots \bigwedge ((x = 0) \vee (y = 0)) \bigwedge \ldots$$

forbids all (full) assignments that contain a forbidden partial assignment that satisfies $(x \neq 0)$ and $(y \neq 0)$.

Because normally we have multiple clauses inside a formula, the question of satisfiability can be turned around to one of unsatisfiability :

- Do the combined forbidden partial assignments of the clauses forbid all assignments?

As we will need to analyze the literals in a clause, when we have a clause

$$C := (a \vee \neg b)$$

of Boolean variables, we say that it contains the following

$$\text{lits}(C) \;=\; \{a, \neg b\}$$

literals. An interesting case appears when the atoms are from NRA. Then we a clause like

$$C := ((xy < 3) \quad \vee \quad \neg(y^2 = 4)$$

contains the literals

$$\text{lits}(C) \;=\; \{(xy < 3), \quad \neg(xy \geq 3), \quad \neg(y^2 = 4), \quad (y^2 \neq 4)\},$$

because the negation in front of an atom can be merged into the polynomial or pulled out. That's why for example the atom $(xy < 3)$ represents the literal of itself and $\neg(xy \geq 3)$ at the same time.

3.1.3 Resolution

A common way to generate new logically entailed formulas is called resolution.

Definition 3.8 (Resolution-step)**.** Given two propositional input clauses

$$(\underbrace{L_1^1 \vee \ldots \vee L_1^1 \vee}_{\text{may be empty}} A) \text{ and } (\neg A \underbrace{\vee L_1^2 \vee \ldots \vee L_m^2}_{\text{may be empty}}),$$

where one contains an atom and the other contains the negation of the same atom, then the clause

$$(L_1^1 \vee \ldots \vee L_n^1 \vee L_1^2 \vee \ldots \vee L_m^2)$$

is called the *resolvent*, that is, the output of a *resolution-step*. This clause

does contain neither A nor $\neg A$, but all the remaining literals of both input clauses.

A resolution-step combines two forbidden partial assignments. For example, the resolution of propositional input clauses

$$(b \vee a) \text{ and } (\neg a \vee \neg c),$$

where the first forbids the partial assignment $[\neg b, \neg a]$ and the second forbids $[a, c]$, produces the resolvent output clause

$$(b \vee \neg c),$$

which forbids the partial assignment $[\neg b, c]$. Intuitively, if $\neg b$ and $\neg a$ are forbidden at the same time by the first input clause, and a and c as well by the second clause, then $\neg b$ and c are forbidden at the same time by the resolvent clause—with no mention of a anymore. In this last forbidden partial assignment a does not need to be mentioned, because no matter how a is assigned, the enlarged assignment will be forbidden by one of the input clauses: If a is assigned false, then the enlarged partial assignment $[\neg b, c, \neg a]$ is forbidden by the first input clause; similarly, it a is assigned true it is forbidden by the second input clause.

The important part about resolution is that if C is a clause that was constructed by one or more resolution-steps from the clauses in a CNF-formula $F(x_1, \ldots, x_n)$, then

$$F(x_1, \ldots, x_n) \models C$$

and

$$\models \quad F(x_1, \ldots, x_n) \Leftrightarrow (F(x_1, \ldots, x_n) \wedge C) \tag{16}$$

hold. In other words, the resolvent clauses follow logically from the input formula and if we add these clauses to the formula, the resulting formula is still equivalent to the original one.

Definition 3.9 (Singleton clause)**.** A clause that contains only a single literal

$$(L),$$

that is, a single atom A or the negation $\neg A$ of one, is called a *singleton clause*

Definition 3.10 (Empty clause)**.** A clause that contains no single literal

$$( )$$

is called the *empty clause*. It represents "false".

If we have two singleton clauses either from the input formula F or derived from one or more resolution-steps,

$$(a) \text{ and } (\neg a),$$

one with an atom and other one with its negation, then using another resolution-step yields the empty clause ( ), which stands for
$false$ (see Def. 3.7), and we have thus proven

$$F(x_1, \ldots, x_n) \models \text{false},$$

which means that the input formula F is unsatisfiable.

3.2 SMT solving techniques

Modern SAT solvers use a variety of sophisticated techniques and heuristics to explore the solution search space of a propositional formula, that is, where the variables accept the Boolean values true and false. The search space consists of the huge but finite number of all possible Boolean variable assignments and those techniques help to avoid enumerating them all. MCSAT enhances some of those techniques to SMT-formulas like QF_NRA-formulas, where the variables accept real number values and thus the search space of all possible real variable assignments becomes infinite.

The main techniques of SAT solvers on propositional formulas, which can be summarized as "guess-and-check", are:

- Decide (a variable's value).*
- Propagate (a variable's value).*
- Analyze (a conflict).*
- (Un)Learn (a lemma).
- Backtrack—take back decisions.

The starred items are points where MCSAT particularly improves existing techniques. We first select a yet-unassigned variable and guess and assign a value for it. This is called a "decision", and there are many heuristics to select the next variable and its value to metaphorically have the highest impact. For example, we can select a variable with the most occurences in clauses so that we can satisfy many clauses with one decision.

Next, we check if a decision implies values for other variables. This is called a "propagation". For example, if we guess and assign a to false for the clause-form formula

$$\underbrace{(a \vee \neg b)}_{1} \quad \wedge \quad \underbrace{(b \vee a)}_{2}, \tag{17}$$

then the first clause forces us to propagate and to assign b to false; we have no choice, otherwise this clause would become false.

Next, after each decision and propagation we check if the currently assigned variables lead to a conflict in some of the clauses, that is, if it makes a clause false. If it does, we determine the decided variables and their values that are actually involved in the conflict, because propagated variables are—by the definition of propagation—assigned values because of our decisions on other variables. We then determine what current (partial) variable assignment is actually forbidden. This is a "conflict analysis". This is another example where the clause structure

is useful, because we can easily recognize a conflict and the forbidden assignment: If our current assignment makes all literals in a clause false, we have a conflict and the clause itself represents the forbidden assignment. However, this assignment could contain propagated variables and there are clever ways to use resolution to find the decided variable culprits and their forbidden asignment. Continuing our example from Eq. 17, having decided a to be false and propagated b to false, leads to a conflict in the second clause: All literals b and a are made false, that is, setting b to false and a to false is a forbidden partial assignment. However, "analyzing" this conflict through resolution would reveal that the actually involved decided variable is a alone and the only actually forbidden partial assignment is setting a to false.

Next, through "analyzing" a conflict we have detected an implicitly forbidden partial assignment that was not directly obvious from the input formula clauses. By using "lemma learning", that is, adding the clause that represent this forbidden assignment to the input formula we can make this implicit assignment explicit. The goal is to not to guess and try any forbidden partial assignment ever again. Intuitively, we can view a clause as an alarm that signals when our decisions have made all of its literals false and we are in conflict. If we have few large clauses with many literals, the signal could come very late after having made lots of decision. The conflict may actually involve only some of our first decisions. Thus, if we add more and shorter clauses, we can detect earlier when we are in conflict. However, there is a trade-off because adding clauses increases the "bookkeeping overhead" and worsens performance, and there are many heuristics when to add a lemma clause but also when to remove one. The added clause C is called a "lemma" because it is a formula that is always logically entailed like a lemma in a mathematical proof:

$$F(x_1, \ldots, x_n) \models C,$$

They don't change the set of satisfying assignments as in Eq. 16. Three types of clauses, that are logically entailed and that we can safely add, are:

- Any clause from the input formula. We normally guess variables and realize that a certain clause from the input formula evaluates to false. Thus all variables that appear in this clause form a forbidden partial assignment and the clause represents it. However, this clause is already part of the input formula.
- Any resolvent clause that was derived through one or more resolution-steps (sef Def. 3.8) from input formula clauses.
- Any valid clause, that is, one that is universally true.

The final technique of modern SAT solvers is called "backtracking". It verbalizes the fact that when we encounter a conflict, we have to take back some of our (guessed) decisions on variable values. There are many heuristics on what decisions to take back.

- If we only ever take back the last decision, this is known as "chronological backtrack".
- If we allow to take back two or more of our last decisions, this is known as "non-chronological backtrack" or more appropiately "backjump", because we "jump back" over many decisions.

- If we take back one or more of our last decisions, but immediately replace them with other decisions, this has—to the best of our knowledge—no name yet.

3.3 MCSAT

MCSAT by Jovanović and de Moura [MJ13] enhances the following previously discussed points from SAT and propositional formulas to SMT-formulas.

- Decide (a variable's value).
- Propagate (a variable's value).
- Analyze (a conflict).

First, in the context of QF_NRA-formulas, MCSAT has two forms of variables:

$$\underbrace{(x^2 + y^2 - 1 > 0)}_{\text{Boolean variable}} \quad \vee \quad \underbrace{(xy - 1 = 0)}_{\text{Boolean variable}}$$

On the one hand we have so-called theory variables—Jovanović and de Moura call them model-variables—that depend on the theory in use. In our case we use the theory of NRA and so our theory variables are real variables like x and y that accept real numbers. On the other hand and independent of any theory, we treat theory atoms like $(xy - 1 = 0)$ as Boolean variables. This means that we can guess and decide values for theory variables like $x \to 0$, and at the same time decide and assert that a theory atom—which we now call a Boolean variable—like $(xy - 1 = 0) \to$ true shall be true.

Second, having Boolean and theory variables allows us to have propagations for the Boolean variables through reasoning on the clause structure—as in SAT—but also through reasoning in the underlying theory.

Third, having Boolean- and theory variables allows us to analyze and resolve conflicts for the Boolean variables through reasoning on the clause structure—as in SAT—but also through reasoning in the underlying theory.

3.3.1 A trail as an ordered, mixed-variable assignment

Jovanović and de Moura call a (partial) mixed-variable assignment, that is, an assignment function that simultaneously assigns values to theory variables and theory atoms (our Boolean variables), a trail.

Definition 3.11 (Trail)**.** A trail is a sequence of Boolean decisions, Boolean propagations, and theory decisions, and has a syntactic form like

$$[L_{\mathrm{D}}, \quad (C_{\mathrm{E}}, L_{\mathrm{P}}), \quad L_{\mathrm{D}}, \quad x \to \text{theory value}, \quad (C_{\mathrm{E}}, L_{\mathrm{P}})].$$

If an element of the trail is a literal L, it represents a Boolean assignment: If L is a theory atom A, we assign A to true; If L is $\neg A$, we assign A to false. For literals, we use subscripts D and P to indicate whether a literal got its value trough a decision of ours or a propagation. In case of a propagation we also store the "explanation", which is a clause C of the input formula—or a logically entailed clause—that forced us to assign the

atom A its value. Finally, if an element of the trail is a theory assignment, well, we have an explicit theory assignment like $x \to 0$.

With a trail M we can evaluate theory atoms, because it represents three assignment functions: First, the Boolean assignment function

$$\vartheta_B(A) = \begin{cases} \text{true} & \text{if } L = A \text{ and } L_{D/P} \in M \\ \text{false} & \text{if } L = \neg A \text{ and } L_{D/P} \in M \\ \text{undef} & \text{otherwise} \end{cases}$$

that can only evaluate theory atoms that appear as decided or propagated Boolean literals in M.

Second, the theory assignment function

$$\vartheta_T(A) = \begin{cases} \text{true} & \text{if } A \text{ evaluates to true after using the theory-assignments} \\ & \text{to replace the free variables in } A \\ \text{false} & \text{if } A \text{ evaluates to false after using the theory-assignments} \\ & \text{to replace the free variables in } A \\ \text{undef} & \text{otherwise} \end{cases}$$

that only uses the theory assignments in M.

Third, the combined assignment function

$$\vartheta_M(A) = \begin{cases} \vartheta_B(A) & \text{if } \vartheta_B(A) \neq \text{undef} \\ \vartheta_T(A) & \text{otherwise} \end{cases}$$

that first evaluates with respect to the Boolean literals in M and only if that is not possible, does it use the theory assignments.

The name "trail" is appropriate, because its represents an assignment as a chronological sequence of our decisions and propagations, where the order is important, for example for backjumps.

For example, a trail like

$$[\neg(x > 0)_{\mathrm{D}}, \quad ((x > 0) \vee (x \leq -1)_{\mathrm{E}},\ (x \leq -1)_{\mathrm{P}}), \quad x \to -1)]$$

represents the mixed-variable assignment that assigns $(x > 0)$ to false, $(x \leq -1)$ to true and x to -1.

There is an interesting case where we can fully evaluate an atom with only the theory assignments even though not all theory variables are assigned. For example with a trail

$$[x \to 0]$$

alone we can evaluate the atom $(x \cdot (y^2 - 3y)z = 0$ to true even though we don't have assigned values to y and z yet.

A trail M can be:

- consistent—For every literal $L \in M$—which implies $\vartheta_B(L) = \text{true}$—the evaluation of $\vartheta_T(L)$ is either true or undef but not false.
- feasible—The literals in M are satisfiable. This implies that if they contain theory variables that are yet unassigned, then there satisfying values for them.

- stable—For every literal $L \in M$ we have $\vartheta_T(L) = \text{true}$, that is, every literal is "supported" or "justified" by the theory assignments. Jovanović and de Moura call such a trail "complete", but the term "stable" is already in widespread use for a similar concept in an SAT variant called Answer Set Programming (ASP).

- complete—There is a theory assignment for every free variable x_i in the input formula $F(x_1, \ldots, x_n)$. Jovanović and de Moura don't use this concept.

Thus, an inconsistent trail M has a literal $L \in M$ where $\vartheta_T(L) = \text{false}$, an infeasible trail has literals in M that taken together are unsatisfiable, an unstable trail has a literal that is not justified by the theory assignments and an incomplete, and in an incomplete trail a theory variable is still unsassigned. The final result of the MCSAT algorithm is then a consistent, feasible, stable and complete trail that satisfies the input formula.

The main task during the construction of this trail is to keep the theory variables and the Boolean variables consistent and feasible in that they don't contradict each other. For example, the trail

$$[(x > 0), \quad x \to 0]$$

is inconsistent and contradicts itself in that the Boolean literal asserts that x is positive, but the theory assignment asserts that x is zero. However, as Jovanović and de Moura point out, consistency on the Boolean variables in a trail does not mean that the trail literals together are satisfiable [JM13, p. 2]. For example, the trail

$$[(x \leq 0), \quad x \to 0, \quad (y \leq 0), \quad (x + y > 0)]$$

is consistent in every literal—because y is yet unassigned—,but the literals together are unsatisfiable, because x and y cannot be both smaller than zero and have a sum that is above zero. A satisfiable trail is called "feasible" to distinguish it in the wording from the satisfiability of the input formula.

3.3.2 The MCSAT algorithm

The MCSAT algorithm by Jovanović and de Moura [MJ13, p. 4,5,6] is given in the form of a transition system:

Boolean-Decide				
$M \parallel F$	$\longrightarrow$	$M,\ L \parallel F$	if	$L \in \mathbb{B}$, $\vartheta_M(L) = \text{undef}$

Boolean-Propagate				
$M \parallel F$	$\longrightarrow$	$M,\ (E, L) \parallel F$	if	$E := (L_1 \vee \ldots \vee L_n \vee L) \in F$, $\forall i\colon \vartheta_M(L_i) = \text{false}$, $\vartheta_M(L) = \text{undef}$

Conflict-Detect				
$M \parallel F$	$\longrightarrow$	$\langle M \parallel F \rangle \vdash C$	if	$C \in F$, $\vartheta_M(C) = \text{false}$

Formula-Satisfiable				
$M \parallel F$	$\longrightarrow$	SAT	if	M satisfies F

Clause-Forget/Unlearn				
$M \parallel F$	$\longrightarrow$	$M \parallel F \setminus \{C\}$	if	$C \in F$, C is learned

Conflict-Resolution-Step				
$\langle M,\ (E, L) \parallel F \rangle \vdash C$	$\longrightarrow$	$\langle M \parallel F \rangle \vdash R$	if	$\neg L \in C$, $R := \text{resolve-step}(E, C)$ removing L

Literal-Consume				
$\langle M,\ (E, L) \parallel F \rangle \vdash C$	$\longrightarrow$	$\langle M \parallel F \rangle \vdash C$	if	$\neg L \notin C$
$\langle M,\ L \parallel F \rangle \vdash C$	$\longrightarrow$	$\langle M \parallel F \rangle \vdash C$	if	$\neg L \notin C$

Trail-Backjump				
$\langle M,\ N \parallel F \rangle \vdash C$	$\longrightarrow$	$M, (E, L) \parallel F$	if	$E := C = (L_1 \vee \ldots \vee L_n \vee L)$, $\forall i\colon \vartheta_M(L_i) = \text{false}$, $\vartheta_M(L) = \text{undef}$, N starts with a Boolean or theory decision,

Formula-Unsat				
$\langle M \parallel F \rangle \vdash \text{false}$	$\longrightarrow$	$UNSAT$		

Clause-Learn				
$\langle M \parallel F \rangle \vdash C$	$\longrightarrow$	$\langle M \parallel F \cup \{C\} \rangle \vdash C$	if	$C \notin F$

Figure 3.1: MCSAT algorithm as a transition system. It transitions from a state $M \parallel F$, which contains a trail M which tries to satisfy formula F in clause-form, to another state. Initially F is the input formula, later it includes learned clauses. $\mathbb{B}$ is a finite basis, $\vdash C$ indicates that a clause C is in conflict, that is, it has been derived and is logically entailed by F but is currently made false by M.

Theory-Propagate

$M \parallel F$	$\longrightarrow$	$M,\ (E, L) \parallel F$	if	$L \in \mathbb{B}$, $\vartheta_M(L) = \text{undef}$, $\text{infeasible}(\llbracket M,\ \neg L \rrbracket)$, $\text{E} := \text{explain}(\llbracket M, \neg L \rrbracket)$

Theory-Decide

$M \parallel F$	$\longrightarrow$	$M,\ x \to \alpha \parallel F$	if	x appears in F, x is unassigned in M, $\text{consistent}(\llbracket M,\ x \to \alpha \rrbracket)$

Theory-Conflict-Detect

$M \parallel F$	$\longrightarrow$	$\langle M,\ x \to \alpha \parallel F \rangle \vdash E$	if	$\text{infeasible}(M)$, $\text{E} := \text{explain}(M)$

Theory-Assignment-Consume

$\langle M,\ x \to \alpha \parallel F \rangle \vdash E$	$\longrightarrow$	$\langle M \parallel F \rangle \vdash E$	if	$\vartheta_M(C) = \text{false}$,

Theory-Assignment-Backjump + Decide

$\langle M,\ x \to \alpha,\ N \parallel F \rangle \vdash C$	$\longrightarrow$	$M,\ L \parallel F$	if	$C = (L_1 \vee \ldots \vee L_m \vee L)$, $\forall i\colon \vartheta_M(L_i) = \text{undef}$, $\vartheta_M(L) = \text{undef}$

Figure 3.2: MCSAT algorithm as a transition system continued.

The transition system that we see in Fig. 3.1 is adapted from Jovanović and de Moura [MJ13, p. 4-6], where you can find an more detailed explanation of all the transition rules. Presenting the MCSAT algorithm as a transition system is useful to hide much of the implementation details but still highlight the development of the trail M, the set of clauses F currently to satisfy, and a possible clause C in conflict.

Among other things, it leaves open:

- Wether to make a theory decision or a Boolean decision.
- Which Boolean variable/theory atom to decide and what value to use.
- Which theory variable to decide and what value.
- When to learn and unlearn a clause.
- What literal to choose from the finite basis $\mathbb{B}$.

These are all points where existing techniques and heuristics can be "plugged in" and that can be adjusted according to experience and depending on the problem domain.

Most notably, an explanation-clause E—when it is generated—is added to the trail but not automatically added to the set of formula clauses F. This is reasonable because for "Boolean Propagate" the clause E is already a part of F, for "Theory Propagate" is may not be that useful yet, and for "Theory-Conflict-Detect" as a conflict-explanations $\vdash E$ we may first have to check if we can further simplify it through resolution-steps.

3.4 Single Cylindric-Algebraic-Cells in Explanations

For this thesis our main point of interest in the MCSAT algorithm is the conflict-generalizing function explain(M) that appears in the transition rule "Theory-Propagate" and "Theory-Conflict-Detect". Notice Jovanovićs and de Moura's actually use a variant explain(M, L) with two arguments in [MJ13], which we can translate to explain($[\![M, L]\!]$)—the authors actually do this in "T-Propagate" [MJ13, p. 6].

In short, this function accepts as input a consistent, but infeasible trail M and returns a clause that is universally true.

We will first present what this formally means for NRA and how CAD can help, and afterwards present a geometrical, more intuitive view.

The fact that a given trail is consistent, but infeasible means that the theory-assignments don't contradict the literals in M, but the literals together are unsatisfiable. Such a trail can exist, when the literals mention a variable that is not assigned yet. Let

$$F_{\text{Lits}} \coloneqq L_1^M \wedge L_2^M \wedge \ldots \wedge L_t^M \tag{18}$$

represents the literals in M—say t many—combined with logical AND ($\wedge$), and let $x_1, \ldots, x_k$ be the variables that have a theory assignment in M and are assigned to real values $\alpha_1, \ldots, \alpha_k$. Then formally this trail M represents the valid logical formula

$$\models \quad F_{\text{Lits}}(\alpha_1, \ldots, \alpha_k, x_{k+1}, \ldots, x_n) \implies \text{false} \tag{19}$$

is valid. Here we highlight the free variables of F_{Lits}. The reason why we present this formula with an implication will become apparent when we explain the use of CAD in this context. Notice how α_1 to α_k are plugged into the first k variables, so that these variables disappear, and that the fact that this formula is valid implies that whatever values we plug in for the remaining variables x_{k+1}, the formula F_{Lits} will evaluate to "false". In other words, the literals of M are currently infeasible under the theory assignments.

We can rewrite this implication formula into an equivalent formula that again mentions all variables $x_1, \ldots, x_k, \ldots x_n$, including x_1 to x_k:

$$\models \quad (\bigwedge_{i=1}^{k} x_i = \alpha_i) \wedge F_{\text{Lits}}(x_1, \ldots, x_n) \implies \text{false}; \tag{20}$$

these variables had disappeared in the implication formula before. Here ($\bigwedge_{i=1}^{k} x_i = \alpha_i$) contraints the first k variables to be point α, and F_{Lits} contraints the remaining variables x_{k+1} to x_n as before. The reason for this rewrite becomes apparent when we continue to analyze the explain function.

3.4.1 Formal requirements on the explanation function's output

The explain-function formally constructs a valid theory formula of the form

$$\models E := (L_1 \vee \ldots \vee L_n), \tag{21}$$

that is, a clause that is univerally true. The validity requirement of the functions output is used to be applicable with any theory, however not all valid theory formulas are useful. It could simply return F_{Lits} from Eq. 20 in clause-form. Instead, the main idea is that

- the explain-function should "generalize" the theory assignments in conflict—geometrically a point in NRA. In NRA this means to extend the conflicting point to a larger set of conflicting points, which in our case will be cylindric algebraic regions called CAD-cells.

3.4.2 Cells for Non-linear Real Arithmetic explanations

In the context of NRA we construct a formula

$$\models F_{\text{cell}}(x_1, \ldots, x_k) \land F_{\text{Lits}}(x_1, \ldots, x_n) \quad \Longrightarrow \quad \text{false} \tag{22}$$

where F_{cell} is the defining formula (see Def. 2.22) of a CAD-cell around point $\alpha \in \mathbb{R}^k$. This formula $F_{\text{cell}}(\beta_1, \ldots, \beta_k)$ evaluates to true for every point $(\beta_1, \ldots, \beta_k, \ldots, \beta_n) \in \mathbb{R}^n$ whose first k components lie inside the cell. Notice that

- the formula F_{cell} in Eq. 22 replaces $(\bigwedge_{i=1}^{k} x_i = \alpha_i)$ in Eq. 20. Herein lies the "generalization".
- F_{cell} constraints the first k variables—x_1 to x_k—to points in the cell $S \subset \mathbb{R}^k$ around point $\alpha \in \mathbb{R}^k$.
- We still need to add F_{Lits} to constraint the remaining variables x_{k+1} to x_n.

Recall that F_{cell} (see Def. 2.22) represents a cell as boundaries for variables x_i, $i = 1, \ldots, k$ as either

$$\underbrace{(x_i = f(x_1, \ldots, x_{i-1}))}_{\text{atom}} \text{ or } \underbrace{(f_{\text{low}}(x_1, \ldots, x_{i-1}) \leq x_i) \land (x_i \leq f_{\text{high}}(x_1, \ldots, x_{i-1}))}_{\text{atom} \land \text{atom}}$$

Thus, it is a series of atoms in the shape of

$$F_{\text{cell}} := A_1 \land A_2 \land \ldots \land A_d. \tag{23}$$

Using the definitions of F_{cell} (Eq. 23) and F_{Lits} (Eq. 18), the formula we construct and return in the explain function is

$$\models (\neg A_1 \lor \neg A_2 \lor \ldots \lor \neg A_d \lor \neg L_1^M \lor \ldots \lor \neg L_t^M), \tag{24}$$

which is the clause-form of the formula in Eq. 22.

The CAD-cell represented in this formula has an intuitive geometric interpretation. If the variables in the literals of the trail M all have theory assignments, then a CAD-cell is the connected, sign-invariant region around the conflicting point represented by the theory assignments. This cell only contains conflicting points. We have already seen a visual example in Fig. 2.26b of Sec. 2.5.2.

In contrast, we have a new situation when one of the polynomial's variables has no theory-assignement. Let's assume that we have the trail

$$M := [\underbrace{\underbrace{(-x^2 - y + 3}_{p_1} < 0)}_{L_1}, \quad x \to \underbrace{0}_{\alpha_1}, \quad \underbrace{\underbrace{(x^2 - y + 1}_{p_2} > 0)}_{L_2}],$$

that has two polynomials inside two literals and one theory assignment for x. Most notably, y is not assigned. So, we make the following observations:

- In Fig. 3.3a we see the region of (x, y) points that satisfies L_1 and in Fig. 3.3b we see the region that satisfies L_2.
- In Fig. 3.3c we see that L_1, L_2 are satisfiable, but there is no satisfying point along the vertical line through α_1. This point α_1 lies on the x-axis.
- In Fig. 3.3d we see a CAD-cell (only) along the x-axis that behaves like α_1. There is no satisfying point along the y for any point inside the cell.

So, visually we create a CAD-cell only along the axes of those variables that have theory assignments. And the CAD-cell keeps the invariant that every point inside the cell behaves like the conflicting point: there is no point along the axes for the unassigned variables which satisfies the trail literals.

3.4.3 OneCell embedding

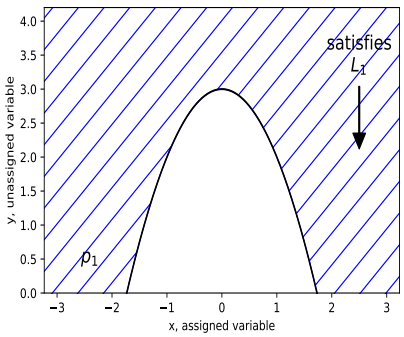
To generalize an infeasible point in the explain-function, the main subtask is to construct a CAD-cell around that point. For this thesis we implemented and use the "OneCell" algorithm by Brown and Košta [BK15], that is decribed in detail Sec. 2.5. This algorithm only accepts polynomials $p(x_1, \ldots, x_k)$ whose free variables have all an assignment represented a point $\alpha := (\alpha_1, \ldots, \alpha_k)$. However the polynomials given by explain(M) in the literals of M may contain additional, unassigned variables $x_{k+1}, \ldots, x_n$ as explained before.

- So, this point α has a lower dimension than the polynomials in M.

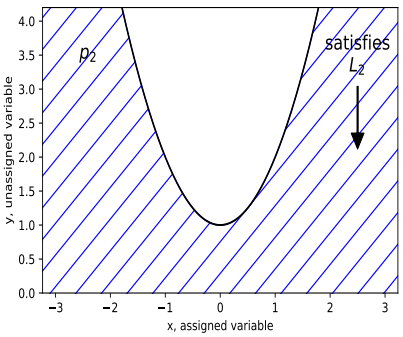
So, OneCell is not directly applicable and the explain-function's main job is to transform its input into a suitable input for the "OneCell" algorithm by removing the unassigned variables. We implemented the explain-function as given in Alg. 1 on top of OneCell. First, it extracts the polynomials from the literals in M into a poly-set P_M. Second, it uses a generic "full" CAD-projection—not OneCell yet— to remove the unassigned variables $x_{k+1}, \ldots, x_n$. In the implementation we use the full the McCallum-Brown projection from Sec. 2.2. A small detail is that we need a variable order for CAD to know the order in which to project and eliminate variables. So, we make sure that we have a variable order like this

$$\underbrace{x_1 \prec x_2 \prec \ldots \prec x_k}_{\text{assigned block}} \prec \underbrace{x_{k+1} \prec \ldots \prec x_{n-1} \prec x_n}_{\text{unassigned block}},$$

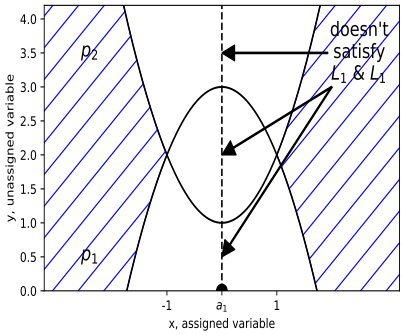
where some variables are assigned in a consecutive block, and some are unassigned—also in a consecutive block. So, a "full" CAD will first remove variables in the "unassigned block" from the polynomials P_M and "OneCell" will construct a cell using the resulting polynomials—these only mention variables from the "assigned block".



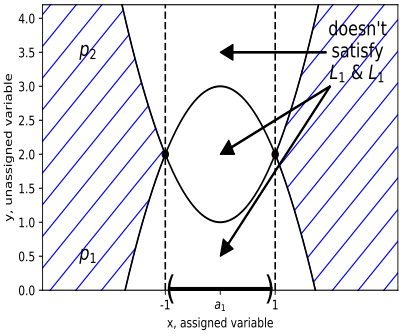
(a) Sign-invariant region that satisfies the literal from M.



(b) Sign-invariant region that satisfies the literal L.



(c) Before: A point $\alpha_1 \in \mathbb{R}^1$ on the x-axis, for which there exists a y-value that satisfies the literal in M and L.



(d) After: A CAD-cell $S \subset \mathbb{R}^1$ along the x-axis such that for every point in it, there still exists a y-value that satisfies the literal in M and L.

Figure 3.3: Qualitive representation of a theory propagation through the generalization of a point to a CAD-cell. We see a contrived polynomial from the single literal in a feasible trail M; we also see the polynomial of a literal L whose negation $\neg L$ would make M infeasible at point α_1 if added. Note that the polynomials have two variables x and y but the point has only one x-component.

Algorithm 1: The explain(M) function.

Input: A trail M (with literals $\{L_1^M, \ldots, L_t^M\}$ and assignments $[x_1 \to \alpha_1, \ldots, x_k \to \alpha_k]$)
Require : A variable order $[x_1 \prec x_2 \prec \ldots \prec x_n]$
Precond: $1 \leq k \leq n$ **and** literals L_i^M are unsatisfiable under assignments (as in Eq. 20).
Output: An explanation clause E

1 Point $\alpha \leftarrow (\alpha_1, \ldots, \alpha_k)$ given by assignments
2 Poly-set $P_M \leftarrow$ polys in literals $\{L_1^M, \ldots, L_t^M, L\}$ from M and L
3 Poly-sets $P_1, \ldots, P_n \leftarrow$ `Nonconst-Irreducible-Factors-Of(`P_M`)` sorted by level 1 to n wrt. variable order
4 **foreach** *level* $i = n$ **to** $k + 1$ **do**
5 $\quad P_1, \ldots, P_{i-1} \leftarrow$ old $P_1, \ldots, P_{i-1}$ buckets with polys of `Full-CAD-Projection(`P_i`)` sorted by level
6 Formula $F_{\text{cell}} \coloneqq \{A_1, \ldots, A_d\} \leftarrow$ `OneCell(`$P_1 \cup \ldots \cup P_k$, α`)`
7 **return** $E \coloneqq (\neg A_1 \vee \neg A_2 \vee \ldots \vee \neg A_d \vee \neg L_1^M \vee \ldots \vee \neg L_t^M \vee L)$

4 Benchmarks

We implemented the "OneCell" CAD algorithm (see Sec. 2.5) as presented by Brown and Košta [BK15] in C++ within the SMT-solving framework SMT-RAT[3], one of the main projects of the I2 institute at the RWTH Aachen university.

This framework uses CArL[4], another main project of the I2 institute, which is an arithmetic toolbox with efficient representions of polynomials, rational numbers and real algebraic numbers. It includes efficiently implemented polynomial operations to compute discriminants, resultants, to isolate roots of a polynomial and to evaluate a polynomial under a given assignment of its variables. The factorization of a polynomial into its irreducible factors—necessary in CAD—is implemented as a wrapper around CoCoA[5], another library with efficient polynomial operations.

The SMT-RAT framework includes an implementation of the MCSAT framework (see Sec. 3.3.2) by de Moura and Jovanović [MJ13] to be used on quantifier-free formualas over the theory of Non-linear Real Arithmetic (QF_NRA). As such, SMT-RAT has an explanation backend that receives a set of conflicting constraints together with a conflicting assignment-point and returns a generalization of the conflicting point to a conflicting region. The existing explanation backend is based on NLSAT [JM13], a predecessor of MCSAT (see Sec. 5.1 for more details), but constructs a single CAD-cell using a "model-based" projection variant that is similar to the OneCell algorithm. In our OneCell implementation we process the input polynomials into the projection factor set in a depth-first search manner to find the smallest bounds for each level that our CAD-cell is forced to have to ensure sign-invariance. The SMT-RAT backend, however, processes the input polynomials into the projection factor set in a breadth-first search manner and finds the smallest bounds level by level, beginning at the top. Both these CAD variants use the same underlying CAD foundation by McCallum [Bro05] that is different from the foundation by Collins [ACM84] which is used by Jovanović and de Moura in their implementation of NLSAT.

We tested our implementation on the complete QF_NRA benchmark set from the SMT-LIB initiative[6] against Z3 and several SMT solving strategies in SMT-RAT.

[3] `https://smtrat.github.io/`
[4] `http://smtrat.github.io/carl/`
[5] `http://cocoa.dima.unige.it/`
[6] `http://smtlib.org/`

The solvers we compared were:

- MCSAT-OC: The MCSAT implementation of SMT-RAT combined with our OneCell implementation as its explanation backend. There is no preprocessing, or other module activated.
- MCSAT-NL: The existing MCSAT implementation of SMT-RAT as described previously. There is no preprocessing, or other module activated.
- MCSAT-PP-OC: As MCSAT-OC, but with input preprocessing to simplify the input formula.
- MCSAT-PP-NL: As MCSAT-NL, but with input preprocessing.
- MCSAT-PP-FM-VS-OC: As MCSAT-OC but with input preprocessing and other activated modules. This strategy first tries to use Fourier-Motzkin (FM) in the explanation backend. This works only for linear constraints. If this backend fails, it tries Virtual Substitution (VS) as another backend. This only works for polynomials with degree of at most 2 in each free variable. If that fails, it tries our OneCell backend. We use these backends in the order of expected performance. In general, FM is faster than VS and VS is faster than OC.
- CAD: The regular SMT-RAT solver that uses CDCL(T)-style SMT solving with an incremental version of CAD as its theory solver.
- Z3[7]: The state-of-the-art solver by Microsoft Research in version 4.6.

We ran each test on a machine with 4 GB RAM and a time limit of 60 seconds. The results can be found in Fig. 4.1 and Fig. 4.2 and were as follows:

In Fig. 4.1 we that MCSAT-OC, the SMT-RAT implementation of MCSAT with our OneCell backend, outperforms MCSAT-NL by proving 100 more instances to be unsatisfiable and 70 more instances to be satisfiable. Similar to MCSAT-NL, we see that MCSAT-OC without any preprocessing or other optimizations is better than CAD, the regular SMT-RAT solver that doesn't use MCSAT, and solves 417 more instances in the same time.

Furthermore, we see that MCSAT-OC, similar to MCSAT-NL, profits from simplifying an input formula with preprocessing, because MCSAT-PP-OC is able to solve 319 more instances. Finally, we see that using OneCell as a fallback backend when Fourier-Motzkin and Virtual Substituion fail, is a further improvement. MCSAT-PP-FM-VS-OC solves 68 more instances than MCSAT-PP-OC. In these instances there must have been calls to the explanation backend where the conflicting core contained only linear constraints and FM was applicable, or the core only contained non-linear polynomials with small degrees so that VS was applicable. However, Z3 solved 10027 instances in total, which is 833 instances more than our best variant MCSAT-PP-FM-VS-OC. We assume that this is due to more efficient code and more optimizations in Z3.

[7] `https://github.com/Z3Prover/z3`

Solver	sat	unsat	solved		runtime	
MCSAT-OC	4327	4478	8805	76.6 %	0.82	
MCSAT-NL	4257	4378	8635	75.2 %	0.96	
MCSAT-PP-OC	4490	4636	9126	79.4 %	0.93	
MCSAT-PP-NL	4426	4575	9001	78.3 %	1.00	
CAD	4362	4026	8388	73.0 %	0.76	
MCSAT-PP-FM-VS-OC	4524	4670	9194	80.0 %	1.00	
Z3 v4.6	4929	5098	10027	87.3 %	0.64	

Figure 4.1: Statistics of solved problem instances. SMT-LIB contains 11489 instances in total.

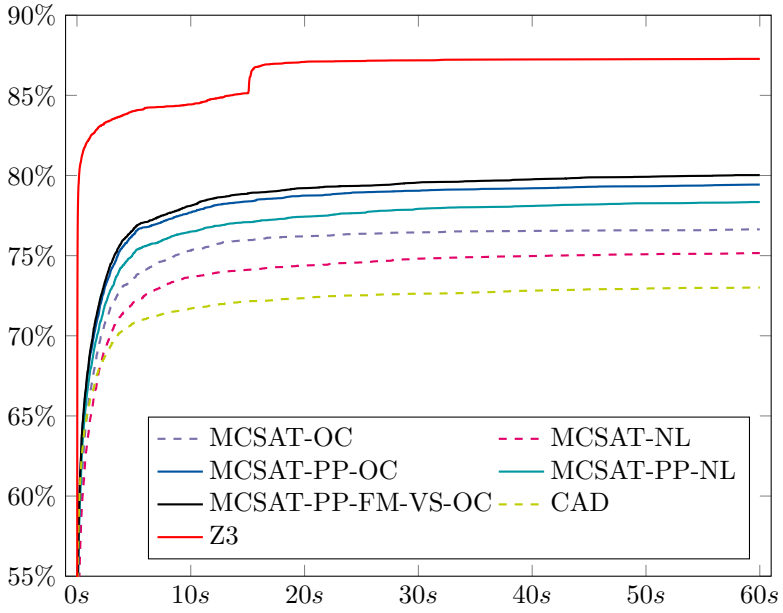


Figure 4.2: Percentage of instances solved within a given time. Note that the y-axis does not begin at 0.

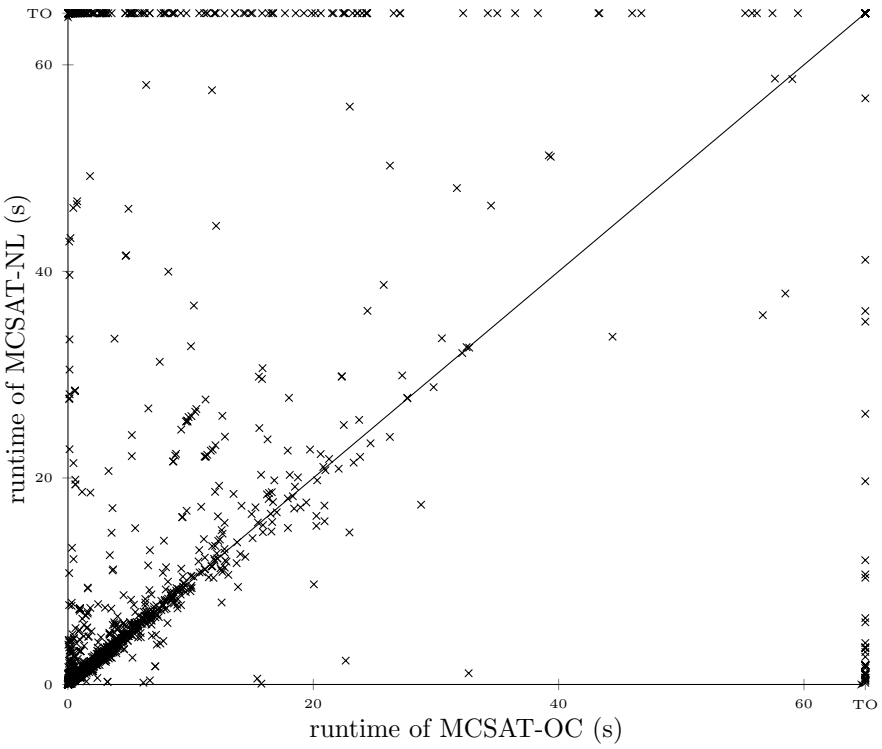


Figure 4.3: Comparison of runtimes between MCSAT-OC and MCSAT-NL on individual instances.

We further analyzed how the existing MCSAT implementation in SMT-RAT, called MCSAT-NL, compared to our OneCell variant MCSAT-OC. In the scatter plot in Fig. 4.3 we see that OneCell has a similar performance on most problem instances (the points along the diagonal line) as MCSAT-NL. There are a few instances where MCSAT-NL is faster (points in the lower triangle), but on much more instances (points in the upper triangle) our OneCell variant is faster; on some instances it can be roughly 20 times as fast and more. In these cases we assume that the larger cells we construct with OneCell exclude more unsatisfying assignments and therefore speed up the assignment space exploration.

Most interestingly, there is a large number of instances that our variant was able to solve very fast, but which MCSAT-NL didn't solve at all because of timeouts (points in the upper left corner). In contrast, there are also instances where our variant ran out of time and where MCSAT-NL was able to solve them quickly (lower right corner). There seems to be an inherent problem structure where the constructed single cells by OneCell are enourmously larger than in MCSAT-NL and therefore speed up the assignment space exploration. However, we also assume that there are cases where a slightly larger or slightly different cell created by OneCell may enable a completely different exploration sequence in MCSAT and may trigger helpful heuristics, which aren't triggered in MCSAT-NL.

5 Related Work

5.1 CAD Foundations

The main two branches of CAD-algorithms are due to Collins and McCallum. The first CAD-algorithm is due to Collins [Col75] [ACM84]. He introduced the algorithm in three phases, projection, base-case and lifting, as described in Sec. 2.2.

Definition 5.1 (Collins-Full-Projector)**.** Given a set of polynomials $P \subset \mathbb{Z}[\boldsymbol{x}_1^n]$ of mixed levels, we call

$$\begin{aligned}\mathrm{PROJ}_{\mathrm{Collins}}(P) := &\bigcup_{p\in P} \mathrm{coeffs}_{x_n}^{\neq 0}(p) \\ &\cup \bigcup_{p\in P} \bigcup_{r\in \mathrm{red}(p)} psc(r, r') \\ &\cup \bigcup_{\substack{p,q\in P \\ p\neq q}} \bigcup_{\substack{r\in \mathrm{red}(p) \\ s\in \mathrm{red}(q)}} psc(r, s)\end{aligned}$$

He introduced a projection operator [ACM84, p. 18] which operates on a set of polynomials of mixed levels, since at that time he didn't have the notion of a polynomial level, at least not in the projection-phase. His projection operator computes for each polynomial the coefficients with respect to the variable x_n, and the "principal subresultant coefficient set" of every of its reducta with the reductum's derivative. Furthermore, for every pair of reducta of two distinct polynomials of the input set, it computes the "principal subresultant coefficient set" of this pair. These are a lot of polynomials.

However, his lifting is straigth-forward, once we categorize his projection factor set into buckets P_1 to P_n of same-level polynomials:

1. D_1—sign-inv on $\mathbb{R}^1$ by base-case-construction using P_1
2. D_2—sign-inv on $\mathbb{R}^2$ by lifting D_1 using P_2
3. ...
4. D_{n-1}—sign-inv on $\mathbb{R}^{n-1}$ by lifting D_{n-2} using P_{n-1}
5. D_n—sign-inv on $\mathbb{R}^n$ by lifting D_{n-1} using P_n

This notation is adapted from [Bro01].

McCallum's CAD algorithm—conceptually all CAD algorithms for that matter—uses the same three phases [McC84] [McC98]. He derived a new projection operator, that produces a much smaller projection factor set than Collin's algorithm. McCallum's work is based on new discoveries in the area of topology by Zariski [Zar75]. However, his algorithm is only complete for a subset of polynomials, called "well-oriented" polynomials (see Def. 2.30). For arbitrary, integral polynomials his algorithm may fail and his lifting-phase is more complicated, because it has to look out for and modify the projection factor set based on "nullifying cells".

Definition 5.2 (McCallum-Full-Projector)**.** Assume we have a finite set P of irreducible polynomials of $\mathbb{Z}[\boldsymbol{x_1^n}]$, each of the same level k with $1 \leq k \leq n$ and let

$$\text{coeffs}(P) \coloneqq \bigcup_{p \in P} \text{coeffs}_{x_k}^{\neq 0}(p), \qquad \text{res}(P) \coloneqq \bigcup_{p,q \in P, p \neq q} \text{res}_{x_k}^{\neq 0}(p,q),$$

$$\text{discr}(P) \coloneqq \bigcup_{\substack{p \in P \\ deg_{x_k}(p) \geq 2}} \text{discr}_{x_k}^{\neq 0}(p),$$

then the set of polynomials

$$\text{PROJ}_{\text{McCallum}}(P) \quad \coloneqq \quad \text{coeffs}(P) \quad \cup \quad \text{discr}(P) \quad \cup \quad \text{res}(P),$$

is called the "projection of P." It contains only polynomials of at most level $k-1$.

Although the projection factor set of McCallums' projection is much smaller, his lifting is more complicated due to the "order-invariance" rather than sign-invariance requirement of the intermediate cells. Furthermore, the possibility of "nullifying cells" requires complicated checks and the modification of the projection factor set after each lifting-step to guarantee or to restore order-invariance.

1. D_1—ord-inv on $\mathbb{R}^1$ by base-case-construction using P_1
2. D_2—ord-inv on $\mathbb{R}^2$ by lifting D_1 using checked and modified P_2
3. ...
4. D_{n-1}—ord-inv on $\mathbb{R}^{n-1}$ by lifting D_{n-2} using checked and modified P_{n-1}
5. D_n—sign-inv on $\mathbb{R}^n$ by lifting D_{n-1} using checked and modified P_n

Given a set of square-free polynomials, McCallum's projector computes the non-zero coefficients and the discriminant of each polynomial and the resultant between any two distinct polynomials of that set [McC84, p. 46] [McC98] [Bro01].

He later discovered that his projection operator could be simplified even further for 3-variable polynomials [McC88], just as Collins [Col75] did for 2-variable polynomials, and extended this discovery [McC98] to polynomials with with any number of variables.

Hong [Hon90] improved Collins's original projection operator [Col75] [ACM84] in reducing the number of polynomials that are required to be computed, but he didn't reduce the number as far as McCallum [McC84] [McC98] did. However, his improvement also didn't restrict his algorithm to certain subsets of polynomials like McCallum's improvements.

5.2 Satisfiability and Satisfiability Modulo Theories

The difference between Satisfiability (SAT) problems and Satisfiability Modulo Theories (SMT) problems is that a problem of the first involves a propositional formula where the atoms—the smallest parts of a formula that can have a truth value—are plain Boolean variables, and a problem of the second involves a quantifier-free, first-order logic formula with atoms over a certain theory like the theory of Non-Linear Real arithmetic (NRA). Hence the part "Modulo Theories".

The main difference lies in the constuction of a conflict-explanation $E := (L_1 \vee \ldots \vee L_n)$. An explanation is a formula in clause-form that was derived to be logically entailed by the input formula —so it must be made true— but is made false by the current variable assignment. In DPPL(T) and CDCDL(T) the atoms in the literals L_i do all previously exist within the input formula. Recall that a clause directly represents a forbidden partial variable assignment and indirectly a set of forbidden full assignments. Roughly, the more forbidden full assignments, the better. So CDCL(T) can't generalize as well as MCSAT when MCSAT is able to exploit the theory reasoning capabilities in its "explain" function. In the case of Non-linear arithmetic, this function can construct single CAD-cells, which generalize a conflict and therefore forbid more assignments than CDCDL(T). However, the representation of a CAD-cell as a formula requires new literals that don't necessarily exist within the input formula.

The Non-Linear SAT framework (NLSAT) and its implementation by Jovanović and de Moura [JM13] is the predecessor of MCSAT, which is basically a specialisation of MCSAT to the theory of Non-Linear Real arithmetic. NLSAT is the inspiring example which we try to improve in our implementation. NLSAT uses a modified Collins-CAD-algorithm (see [JM13, p. 347ff]), which constructs single, generalizing CAD cells around infeasible points using what they call a "model-based Collins projection operator" (see [JM13, p. 349]). It exploits the fact that we only construct a single cell. The infeasible point is represented by a real-variable assignment, which can become a "model" if we view SMT-solving from a theorem-proving perspective, hence the name "model-based". Their modified CAD-algorithm produces a smaller "projection factor set" (see Sec. 2.2) and is therefore faster than the regular Collins-CAD-algorithm to construct a single cell.

In the same spirit as Jovanović and de Moura's modified Collins-CAD-algorithm, we use the modified McCallum-Brown-CAD-algorithm called "One-Cell" by Brown and Košta [BK15]. We use the OneCell-algorithm in MCSAT in the same way as Jovanović and de Moura used their CAD variant, that is, to construct generalizing CAD-cells around infeasible points. Brown and Košta's modified CAD-algorithm also exploits the fact that we only construct a single cell around a point, and is therefore it is equally "model-based". However, this modified CAD-algorithm theoretically produces a smaller projection factor set than the one used by Jovanović and de Moura, and therefore should produce larger cells in less time.

6 Conclusion

We have successfully implemented the OneCell algorithm by Brown and Košta [BK15] and have embedded it into the MCSAT implementation of the SMT-RAT framework to solve quantifier-free formulas over the theory of Non-linear Real Arithmetic (NRA). This MCSAT implementation is based on a publication by Jovanović and de Moura [MJ13].

We have shown that our OneCell embedding is not only theoretically, but also practically better: On its own it slightly outperforms the existing SMT-RAT variation of MCSAT that is based on de Moura and Jovanović's initial implementation of MCSAT for NRA, called NLSAT [JM13]. On the other hand, it largely outperforms the SMT-RAT variant that is not based on MCSAT, highlighting the fact that reasoning in the clause structure of a formula combined with reasoning over the theory of NRA, as it's done in MCSAT, can be very beneficial.

As a possible line of future research, we would like follow the suggestion of Brown and Košta. In one of their publications they suggest, as a possible improvement of their OneCell CAD-algorithm, that we try to revise the algorithm to merge two CAD-cells into one instead of the current way of merging one polynomial into a single cell [BK15]. This variant would be interesting to us, because it would enable us to parallelize the single cell construction. and this improve the parallel SMT solving speed: We could construct a cell on one processor core for one part of the input polynomials and another cell for the remaing polynomials on another processor core. Afterwards we would merge the two emerging cell into one. Finally, another interesting improvement would be to combine OneCell with the work of McCallum on exploiting equational constraints [McC01] to improve the solving speed of MCSAT on NRA even further.

References

[Col75] George E Collins. "Quantifier elimination for real closed fields by cylindrical algebraic decompostion". In: *Automata Theory and Formal Languages 2nd GI Conference Kaiserslautern, May 20–23, 1975*. Springer. 1975, pp. 134–183.

[Zar75] Oscar Zariski. "On Equimultiple Subvarieties of Algebroid Hypersurfaces". In: *Proceedings of the National Academy of Sciences* 72.4 (1975), pp. 1425–1426. ISSN: 0027-8424. DOI: 10.1073/pnas.72.4.1425. eprint: http://www.pnas.org/content/72/4/1425.full.pdf. URL: http://www.pnas.org/content/72/4/1425.

[ACM84] Dennis S. Arnon, George E. Collins, and Scott McCallum. "Cylindrical Algebraic Decomposition I: The Basic Algorithm". In: *SIAM Journal on Computing* 13.4 (1984), pp. 865–877. DOI: 10.1137/0213054. eprint: https://doi.org/10.1137/0213054. URL: https://doi.org/10.1137/0213054.

[McC84] Scott McCallum. "An Improved Projection Operation for Cylindrical Algebraic Decomposition (Computer Algebra, Geometry, Algorithms)". AAI8500835. PhD thesis. 1984.

[McC88] Scott McCallum. "An improved projection operation for cylindrical algebraic decomposition of three-dimensional space". In: *Journal of Symbolic Computation* 5.1 (1988), pp. 141–161. ISSN: 0747-7171. DOI: https://doi.org/10.1016/S0747-7171(88)80010-5. URL: http://www.sciencedirect.com/science/article/pii/S0747717188800105.

[Hon90] H. Hong. "An Improvement of the Projection Operator in Cylindrical Algebraic Decomposition". In: *Proceedings of the International Symposium on Symbolic and Algebraic Computation*. ISSAC '90. Tokyo, Japan: ACM, 1990, pp. 261–264. ISBN: 0-201-54892-5. DOI: 10.1145/96877.96943. URL: http://doi.acm.org/10.1145/96877.96943.

[GCL92] Keith O. Geddes, Stephen R. Czapor, and George Labahn. *Algorithms for Computer Algebra*. Norwell, MA, USA: Kluwer Academic Publishers, 1992. ISBN: 0-7923-9259-0.

[McC98] Scott McCallum. "An Improved Projection Operation for Cylindrical Algebraic Decomposition". In: *Quantifier Elimination and Cylindrical Algebraic Decomposition*. Ed. by Bob F. Caviness and Jeremy R. Johnson. Vienna: Springer Vienna, 1998, pp. 242–268. ISBN: 978-3-7091-9459-1.

[Duc00] Lionel Ducos. "Optimizations of the subresultant algorithm". In: *Journal of Pure and Applied Algebra* 145.2 (2000), pp. 149–163. ISSN: 0022-4049. DOI: https://doi.org/10.1016/S0022-4049(98)00081-4. URL: http://www.sciencedirect.com/science/article/pii/S0022404998000814.

[Bro01] Christopher W. Brown. "Improved Projection for Cylindrical Algebraic Decomposition". In: *Journal of Symbolic Computation* 32.5 (2001), pp. 447–465. ISSN: 0747-7171. DOI: https://doi.org/10.1006/jsco.2001.0463. URL: http://www.sciencedirect.com/science/article/pii/S0747717101904638.

[McC01] Scott McCallum. "On propagation of equational constraints in CAD-based quantifier elimination". In: *Proceedings of the 2001 international symposium on Symbolic and algebraic computation*. ACM. 2001, pp. 223–231.

[Coh03] Joel S Cohen. *Computer algebra and symbolic computation: Mathematical methods*. Universities Press, 2003.

[Bro05] Christopher W. Brown. *The McCallum Projection, Lifting, and Order-Invariance*. Tech. rep. NAVAL ACADEMY ANNAPOLIS MD DEPT OF COMPUTER SCIENCE, 2005.

[Con+05] Evelyne Contejean et al. "Mechanically proving termination using polynomial interpretations". In: *Journal of Automated Reasoning* 34.4 (2005), p. 325.

[BD07] Christopher W. Brown and James H. Davenport. "The Complexity of Quantifier Elimination and Cylindrical Algebraic Decomposition". In: *Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation*. ISSAC '07. Waterloo, Ontario, Canada: ACM, 2007, pp. 54–60. ISBN: 978-1-59593-743-8. DOI: 10.1145/1277548.1277557. URL: http://doi.acm.org/10.1145/1277548.1277557.

[BHM09] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*. Vol. 185. IOS press, 2009.

[Bro13] Christopher W. Brown. "Constructing a Single Open Cell in a Cylindrical Algebraic Decomposition". In: *Proceedings of the 38th International Symposium on Symbolic and Algebraic Computation*. ISSAC '13. Boston, Maine, USA: ACM, 2013, pp. 133–140. ISBN: 978-1-4503-2059-7. DOI: 10.1145/2465506.2465952. URL: http://doi.acm.org/10.1145/2465506.2465952.

[JM13] Dejan Jovanović and Leonardo de Moura. "Solving Non-linear Arithmetic". In: *ACM Commun. Comput. Algebra* 46.3/4 (Jan. 2013), pp. 104–105. ISSN: 1932-2240. DOI: 10.1145/2429135.2429155. URL: http://doi.acm.org/10.1145/2429135.2429155.

[MJ13] Leonardo de Moura and Dejan Jovanović. "A Model-Constructing Satisfiability Calculus". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–12. ISBN: 978-3-642-35873-9.

[BK15] Christopher W. Brown and Marek Košta. "Constructing a single cell in cylindrical algebraic decomposition". In: *Journal of Symbolic Computation* 70 (2015), pp. 14–48. ISSN: 0747-7171. DOI: https://doi.org/10.1016/j.jsc.2014.09.024. URL: http://www.sciencedirect.com/science/article/pii/S0747717114000923.