





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

MASTER OF SCIENCE THESIS

# FORMALIZING A PROOF SYSTEM FOR SINGLE CELL CONSTRUCTION IN LEAN

**Carsten Peter Perkampus**

*Communicated by*
Prof. Dr. Erika Ábrahám

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

*Additional Advisors:*
Dr. Jasper Nalbach
Valentin Promies

Aachen, October 16, 2026

**Abstract**

Cylindrical algebraic decomposition (CAD) is a fundamental technique for reasoning about real algebraic constraints, but its algorithms rely on complex symbolic computations and heuristic choices that make full formal verification challenging. This thesis investigates a certificate-based approach to checking derivations in the proof system for levelwise single-cell construction in the Lean theorem prover.

The work formalizes the central objects and properties of the proof system. A substantial part of the rule system is represented in Lean, with elementary rules proved directly and more involved rules stated as axioms where the underlying mathematics has not yet been formalized in Lean.

To avoid reimplementing the CAD proof search inside Lean, the thesis separates proof generation from proof checking. An external procedure may construct a CAD proof using its existing algorithms and heuristics and export the resulting derivation as a certificate. A proof-replay mechanism based on an Alethe-inspired certificate syntax reconstructs this derivation in Lean by instantiating the formalized inference rules. A domain-specific tactic checks selected mathematical side conditions required for these rule applications.

The prototype shows that the proof system for levelwise single-cell construction can be used as an interface for proof-producing real-algebraic reasoning. The current implementation does not yet verify the complete proof system or arbitrary CAD certificates, but it already provides the main components of a checker: formal rule interfaces, certificate replay, and checking selected side conditions. Future work includes completing the remaining mathematical proofs, developing general root-isolation support, extending the automatic verification of algebraic side conditions, and integrating the checker with a practical CAD implementation.

# Contents

**1 Introduction** **9**
1.1 Research Questions . . . . . 11
1.2 Related Work . . . . . 13
1.3 Thesis Outline . . . . . 14

**2 Preliminaries** **15**
2.1 Algebraic Foundations . . . . . 15
2.2 Cylindrical Algebraic Decomposition . . . . . 22
2.3 Levelwise Single-Cell Construction . . . . . 25
2.4 Lean . . . . . 35
2.5 Alethe . . . . . 41

**3 Formalization** **43**
3.1 Core Representations . . . . . 43
3.2 Formalization of Cell Properties . . . . . 46
3.3 From Inference Rules to Lean Theorems . . . . . 49
3.4 Formalization Boundary . . . . . 51

**4 Proof Automation** **53**
4.1 Proof Certificates . . . . . 53
4.2 Certificate Replay . . . . . 54
4.3 Example: Verifying a Nonlinear Covering . . . . . 54
4.4 Verification of Certificate Leaves . . . . . 61
4.5 Trust Boundary and Current Limitations . . . . . 64

**5 Conclusion** **65**
5.1 Summary . . . . . 65
5.2 Future Work . . . . . 66

**Bibliography** **71**

# Chapter 1

# Introduction

*Nonlinear real arithmetic* (NRA) concerns polynomial equations and inequalities over real-valued variables, including nonlinear expressions involving products of variables or higher powers. A central problem is deciding whether a Boolean combination of such constraints has a solution over the real numbers. Decision procedures for NRA are an important component of automated reasoning systems and are used, for example, in satisfiability modulo theories, quantifier elimination, verification, and symbolic computation. A classical method for deciding such problems is *Cylindrical Algebraic Decomposition* (CAD), originally introduced by Collins for quantifier elimination over the real numbers [Col75]. CAD decomposes $\mathbb{R}^n$, where $n$ is the number of variables, into finitely many regions, called *cells*, on which a given collection of polynomials exhibits invariant behaviour, for example a constant sign. This decomposition can subsequently be used to determine the truth of polynomial constraints throughout these regions.

Despite its generality, classical CAD is computationally expensive. In particular, constructing a complete decomposition of the entire search space may perform considerably more work than is necessary for deciding a particular formula. Modern approaches therefore increasingly employ local or exploration-guided variants of CAD [ÁDEK21, NÁS+24]. Instead of constructing a full decomposition in advance, these methods construct only those cells that are required during the search. Such techniques occur, for example, as part of the Model Constructing Satisfiability Calculus (MCSAT) [JBd13]. While classical CAD constructs a complete decomposition into pairwise disjoint cells, *levelwise single-cell construction* [NÁS+24] constructs an individual cylindrical cell around a given sample point. *Cylindrical algebraic coverings* [ÁDEK21] combine such local cells to cover a region, without requiring them to be pairwise disjoint.

The increased flexibility of local CAD algorithms, however, also complicates their correctness arguments. A traditional CAD *projection operator* prescribes a fixed collection of projection polynomials whose invariance guarantees the correctness of the subsequent lifting process. In a local setting, several different sets of conditions may instead be sufficient to establish the desired property of a particular cell. Depending on its location and the surrounding polynomial roots, certain resultants, discriminants, or coefficients may be irrelevant, while alternative projection choices can lead to larger cells or cheaper computations. The correctness argument is therefore no longer naturally tied to a single deterministic projection operation.

To address this issue, previous work introduced a *proof system* for levelwise single-cell construction [NÁS+24]. Rather than expressing projection as a fixed transformation on sets of polynomials, the proof system describes properties of the cell under construction and provides inference rules that reduce these properties to simpler obligations. Properties express, among other things, that a sample point is contained in a cell, that a symbolic interval correctly represents a cell boundary, that polynomial roots preserve a particular ordering, or that a collection of cells covers a complete cylinder [Nal25]. A CAD-based solving procedure can then be understood as a high-level *proof strategy*: it selects suitable cell representations and repeatedly applies inference rules until the resulting obligations can be established at lower dimensions.

This separation also makes the proof system useful for formal verification. A run of a CAD-based solver can be viewed as constructing a derivation while it searches. Instead of verifying the complete solver implementation, we can let the solver output the relevant proof-rule applications and check the resulting *certificate* independently. Similar proof-reconstruction architectures are used in other areas of automated reasoning [MMK+25].

Formalizing this idea in Lean requires us to make several details explicit that are usually left implicit on paper. The domains and dimensions of polynomials, sample points, indexed roots, and symbolic intervals must be represented in their types or in additional well-formedness conditions. Applying a proof rule to a concrete certificate also creates *side conditions*, for example about the existence of indexed roots, relations between interval bounds, polynomial identities, and arithmetic constraints. A practical checker therefore needs both a formal representation of the proof system and automation for these instances.

This thesis investigates the formalization of the CAD proof system in the interactive theorem prover Lean. The objective is to establish a foundation on which proof traces generated by CAD-based reasoning procedures can be represented and checked by Lean's kernel. To this end, the mathematical objects in the proof system are translated into typed Lean definitions, selected inference rules are expressed as machine-checkable theorems, and automation is developed for proving side conditions arising during their application.

A central challenge of the formalization concerns the representation of indexed root expressions and symbolic cell descriptions. In the mathematical proof system, an expression referring to the $k$-th root of a polynomial can be used as a symbolic description of a cell boundary. Its interpretation depends both on the defining polynomial and on the point over which its roots are considered. Similarly, symbolic intervals may contain finite root expressions or infinite bounds and may represent sections or sectors of a CAD. The formalization must be able to represent these semantics while retaining enough structure to conveniently instantiate the inference rules of the proof system.

Root relationships provide another important part of the formal model. Local CAD constructions frequently rely on statements asserting that one root function remains below another throughout an underlying cell. The proof system records such information using indexed root orderings and connects these orderings to algebraic conditions involving resultants and delineability. Representing these orderings explicitly allows the formalization to mirror the local nature of the original proof system instead of reverting to a global CAD construction.

Beyond the encoding of the mathematical concepts themselves, this thesis considers the practical problem of proof automation. Concrete applications of the inference

rules contain many obligations whose proofs are conceptually straightforward but would make generated certificates impractical if they required manual interaction. The developed automation therefore attempts to recognize common goals arising from CAD certificates and reduce them to simpler algebraic or logical statements that Lean can decide or prove automatically.

In this thesis, we use Lean to check proofs produced by CAD-based search procedures. We do not attempt to reimplement the complete CAD search inside Lean; CAD optimizations can remain outside Lean as long as their results can be justified by verified proof rules.

More concretely, the thesis makes the following contributions. First, it develops a Lean representation of the central objects required for expressing applications of the CAD proof system, including sample points, indexed root expressions, symbolic intervals, and root-ordering information. Second, it formalizes the semantics of relevant cell properties and connects them to corresponding inference rules from the mathematical proof system. Third, it investigates the formal verification of selected proof rules and makes their assumptions and dependencies explicit in Lean. Fourth, it develops domain-specific proof automation for recurring obligations that arise when concrete instances of these rules are checked. Finally, the formalization is evaluated on representative CAD proof scenarios, including cell descriptions and coverings involving nonlinear polynomial constraints. These examples demonstrate how a high-level CAD derivation can be represented as a collection of proof obligations and checked within the formal framework.

An additional objective of the thesis is to identify the *trusted boundary* of such a proof-producing approach. Formalizing an existing mathematical proof system makes assumptions explicit that can easily remain implicit in a paper implementation. Some results may follow directly from existing mathematical libraries, while others depend on substantial theories of real algebra, topology, or polynomial root behaviour that are not yet available in the required form. Distinguishing machine-verified results from remaining assumptions is therefore itself an important outcome of the formalization. In the current development, the components still missing for a fully verified proof checker for CAD-based reasoning are mainly results on delineability, the preservation of root orderings over a cell, and geometric properties of cells such as connectedness.

## 1.1 Research Questions

As part of this thesis, we aim to answer the following research questions:

#### RQ1: How can the CAD proof system be formalized in Lean?

The CAD proof system relies on mathematical objects such as sample points, multivariate polynomials, indexed root expressions, symbolic intervals, and orderings of polynomial root functions. While these objects can often be handled implicitly in a mathematical presentation, a formalization in Lean requires their types, dimensional dependencies, well-formedness conditions, and semantics to be represented explicitly. This question investigates how the objects and properties of the CAD proof system can be represented in Lean while preserving their intended mathematical meaning. In particular, it considers how indexed root expressions, symbolic cell boundaries, projections between dimensions, and root orderings can be encoded in a way that is

both mathematically faithful and suitable for the subsequent formalization of inference rules.

**RQ2: To what extent can the inference rules of the CAD proof system be formally verified in Lean?**

The CAD proof system reduces properties of a cell to simpler properties through a collection of inference rules. For a formally verified proof checker, these rules must ultimately be represented as machine-checked theorems whose conclusions follow from their stated assumptions. This question investigates how the mathematical inference rules can be translated into Lean and which mathematical prerequisites are required to prove them. It further aims to identify where the existing Lean and Mathlib libraries provide sufficient results and where additional formalization of real algebra, polynomial root behaviour, topology, or related concepts is necessary. An important aspect of this question is therefore to determine the trusted boundary of the current formalization by distinguishing formally verified rules from rules or supporting mathematical results that still depend on additional assumptions.

**RQ3: How can the proof obligations from concrete applications of the CAD proof system be automated?**

Even when an inference rule has been formalized as a theorem, applying it to a concrete CAD proof produces numerous side conditions. These include, for example, statements about indexed root expressions, symbolic interval bounds, sample points, polynomial identities, and relations between roots. Proving all such obligations manually would make the use of the formalized proof system impractical as a certificate checker. This question therefore investigates which recurring proof obligations can be identified and automatically discharged. In particular, it considers how custom automation can reduce CAD-specific goals to simpler algebraic, arithmetic, or logical statements that Lean can prove automatically. The question further examines the limits of this automation and the classes of obligations that still require explicit proofs or additional mathematical infrastructure.

**RQ4: How well can the formalization be used to check proofs produced by CAD-based algorithms?**

This question evaluates how well the developed formalization supports checking certificates that consist of applications of proof-system rules. It considers whether representative cell constructions and coverings can be expressed and checked using the formalized objects, inference rules, and proof automation, which parts of the resulting certificates can already be verified end-to-end, and which remaining assumptions or unsupported proof obligations currently prevent a fully verified certificate checker.

## 1.2 Related Work

Cylindrical algebraic decomposition was originally introduced by Collins as a method for quantifier elimination over the reals [Col75]. Subsequent work developed improved projection operators [McC98, Bro01]. More recent CAD-based methods increasingly avoid constructing a complete decomposition when only local information is required. Cylindrical algebraic coverings and levelwise single-cell construction follow this direction by constructing regions around selected sample points and deriving only those projection conditions required for the current search [ÁDEK21, NÁS+24]. The latter

additionally introduces the proof system that forms the mathematical basis of this thesis.

*Proof reconstruction* has also been used to connect automated solvers with interactive theorem provers. Instead of verifying the solver itself, an external solver can produce a proof that is reconstructed or checked inside the theorem prover. The *lean-smt* project follows this approach for SMT solving in Lean [MMK+25]: the SMT solver performs the search and Lean reconstructs the generated proof.

The present work applies this idea to the proof system for levelwise single-cell construction. Its certificates consist of applications of the formalized CAD inference rules and are written in an Alethe-inspired syntax [SFBF21]. This syntax reuses the command-oriented structure of Alethe while allowing CAD-specific rule identifiers to refer directly to Lean declarations. In contrast to *lean-smt*, the focus here is therefore not on reconstructing general SMT reasoning, but on providing a proof-producing interface for the reasoning performed by levelwise single-cell construction.

## 1.3 Thesis Outline

The remainder of this thesis is structured as follows.

Chapter 2 introduces the preliminaries required for the remainder of the thesis. It presents the relevant algebraic foundations and cylindrical algebraic decomposition, with particular emphasis on exploration-guided CAD and levelwise single-cell construction. It then introduces the CAD proof system on which the formalization is based, followed by the Lean theorem prover and the Alethe proof format used as background for the certificate representation.

Chapter 3 presents the formalization of the CAD proof system in Lean. It discusses the representation of the central mathematical objects, including multivariate polynomials, sample points, indexed root expressions, symbolic intervals, and indexed root orderings. Based on these representations, the relevant cell properties and inference rules are translated into Lean declarations. The chapter also identifies the formalization boundary by distinguishing fully proved results from axiomatized rules, admitted auxiliary results, and simplified semantic notions.

Chapter 4 develops the proof-automation infrastructure used to check concrete applications of the formalized proof system. It introduces the certificate representation and replay mechanism, illustrates their interaction on a nonlinear covering example, and presents a tactic for verifying concrete certificate leaves. The chapter concludes by making explicit the trusted boundary and current limitations of the resulting prototype.

Chapter 5 concludes the thesis by summarizing what can currently be represented, verified, and automated within the developed framework. It assesses the extent to which the formalization provides a basis for a proof checker for proof-producing CAD procedures and outlines the mathematical and implementation work required to extend the prototype towards fully verified certificate checking.

# Chapter 2

# Preliminaries

## 2.1 Algebraic Foundations

Cylindrical algebraic decomposition reasons about the signs and roots of multivariate polynomials over the real numbers. This section introduces the algebraic concepts required for the subsequent presentation of CAD, levelwise single-cell construction, and the CAD proof system. In particular, we introduce multivariate polynomials under a fixed variable ordering, real roots, resultants and discriminants, as well as sign invariance, order invariance, and delineability.

### 2.1.1 Multivariate Polynomials

Throughout this thesis, we consider multivariate polynomials with rational coefficients

$$p \in \mathbb{Q}[x_1, \ldots, x_n]$$

under a fixed *variable ordering*

$$x_1 \prec x_2 \prec \cdots \prec x_n.$$

The variable ordering is fundamental to CAD, since projection eliminates variables in reverse variable order, starting with the highest variable, whereas lifting reconstructs higher-dimensional information in the opposite direction [Col75, McC98].

**Definition 2.1.1** (Level of a Polynomial [Nal25])**.** *Let $p \in \mathbb{Q}[x_1, \ldots, x_n]$ be a non-constant polynomial. The* level *of $p$ is the index of the greatest variable occurring in $p$,*

$$\operatorname{level}(p) = \max\{i \mid x_i \text{ occurs in } p\}.$$

*A polynomial $p$ with $\operatorname{level}(p) = i$ is called a* polynomial of level $i$*. Constant polynomials are of level* 0*.*

A polynomial $p$ of level $i$ can be regarded as a univariate polynomial in its highest variable $x_i$, whose coefficients are polynomials in the lower variables:

$$p = c_m x_i^m + c_{m-1} x_i^{m-1} + \cdots + c_1 x_i + c_0,$$

where

$$c_0, \ldots, c_m \in \mathbb{Q}[x_1, \ldots, x_{i-1}].$$

We write $\deg_{x_i}(p) = m$ for the *degree* of $p$ with respect to $x_i$ and $\mathrm{ldcf}_{x_i}(p) = c_m$ for its *leading coefficient* with respect to $x_i$.

**Definition 2.1.2** (Nullification [Nal25])**.** *Let*

$$p = c_m x_i^m + \cdots + c_0 \in \mathbb{Q}[x_1, \ldots, x_i]$$

*with* $c_0, \ldots, c_m \in \mathbb{Q}[x_1, \ldots, x_{i-1}]$*, and let* $s \in \mathbb{R}^{i-1}$*. The polynomial* $p$ *is* nullified at $s$ *if*

$$p(s, x_i) = 0,$$

*or, equivalently, if*

$$c_j(s) = 0 \qquad \text{for every } j \in \{0, \ldots, m\}.$$

**Example 2.1.1.** *Consider the variable ordering*

$$x_1 \prec x_2 \prec x_3$$

*and the polynomials*

$$p_1 = x_1^2 - 2, \qquad p_2 = x_2^2 + x_1 x_2 - 1, \qquad p_3 = x_1 x_3 + x_2.$$

*Then*

$$\mathrm{level}(p_1) = 1, \qquad \mathrm{level}(p_2) = 2, \qquad \mathrm{level}(p_3) = 3.$$

*For*

$$r = x_1 x_2 + x_1 = x_1(x_2 + 1),$$

*substituting* $x_1 = 0$ *yields*

$$r(0, x_2) = 0.$$

*Hence,* $r$ *is nullified at* $x_1 = 0$*.*

### 2.1.2 Real Roots and Root Functions

**Definition 2.1.3** (Real Roots [Nal25])**.** *Let* $p \in \mathbb{R}[x]$ *be a non-zero univariate polynomial. The set of its* distinct real roots *is denoted by*

$$\mathrm{realRoots}(p) = \{r \in \mathbb{R} \mid p(r) = 0\}.$$

*Since* $p$ *has only finitely many roots, the number* $k = |\,\mathrm{realRoots}(p)|$ *of its distinct real roots is finite. If* $k > 0$*, we denote these roots in strictly increasing order by*

$$r_1 < r_2 < \cdots < r_k,$$

*where* $r_j$ *denotes the* $j$*-th smallest distinct real root of* $p$ *for* $1 \leq j \leq k$*.*

For a multivariate polynomial, its roots with respect to the highest variable generally depend on the values of the lower variables. Consequently, when the lower-dimensional point varies, a particular real root may itself vary continuously.

**Definition 2.1.4** (Real Root Function [Nal25])**.** *Let*

$$p \in \mathbb{Q}[x_1, \ldots, x_{i+1}] \qquad \text{and} \qquad R \subseteq \mathbb{R}^i.$$

*A continuous function*

$$\theta : R \to \mathbb{R}$$

*is a* real root function *of $p$ on $R$ if*

$$p(r,\theta(r)) = 0 \qquad \text{for every } r \in R.$$

*Moreover, any other continuous function $\theta' : R \to \mathbb{R}$ satisfying $p(r,\theta'(r)) = 0$ for all $r \in R$ must agree with $\theta$ on all of $R$ if they agree at some point of $R$.*

The graph of a real root function,

$$\{(r,\theta(r)) \mid r \in R\} \subseteq \mathbb{R}^{i+1},$$

can therefore be interpreted as a root of $p$ that moves as the lower-dimensional coordinates vary.

The distinction between a root at one particular sample point and a root function over an entire region is important for CAD. Knowing that $p(s,x_{i+1})$ has a $k$-th real root only provides information at $s$. To use this root as a symbolic boundary of a cell, it must be possible to consistently follow the corresponding root while the lower-dimensional coordinates vary.

**Example 2.1.2.** *Consider*

$$p(x_1,x_2) = x_2^2 - x_1$$

*over the region*

$$R = (0,\infty).$$

*For every $x_1 \in R$, the polynomial*

$$p(x_1,x_2) = x_2^2 - x_1$$

*has the two real roots*

$$-\sqrt{x_1} \qquad \text{and} \qquad \sqrt{x_1}.$$

*These roots are described by the continuous functions*

$$\theta_1(x_1) = -\sqrt{x_1}, \qquad \theta_2(x_1) = \sqrt{x_1}.$$

*At $x_1 = 0$, however, both roots meet at $x_2 = 0$. Thus, extending the region to include or cross $x_1 = 0$ changes the root structure. CAD projection polynomials are used to identify precisely such critical points.*

The levelwise single-cell construction later uses *indexed root expressions* to refer symbolically to roots such as the first or second real root of a polynomial.

### 2.1.3 Resultants and Discriminants

Resultants and discriminants provide algebraic conditions for detecting critical interactions between polynomial roots. Importantly for CAD, eliminating the highest variable produces a polynomial in the remaining lower variables.

**Definition 2.1.5** (Resultant [Nal25]). *Let $p,q \in K[x]$ be univariate polynomials over a field $K$. The* resultant *of $p$ and $q$ with respect to $x$, denoted by*

$$\mathrm{res}_x(p,q),$$

*is the determinant of the Sylvester matrix of $p$ and $q$.*

*If $p$ and $q$ are non-zero, then*

$$\mathrm{res}_x(p,q) = 0$$

*if and only if $p$ and $q$ have a common root over an algebraic closure of $K$.*

For

$$p,q \in \mathbb{Q}[x_1,\ldots,x_i],$$

we regard both polynomials as univariate in $x_i$. Their resultant

$$\mathrm{res}_{x_i}(p,q) \in \mathbb{Q}[x_1,\ldots,x_{i-1}]$$

therefore eliminates $x_i$.

Geometrically, the zeros of the resultant contain those lower-dimensional points above which roots of $p$ and $q$ intersect. Suppose two continuous root functions $\theta_p$ and $\theta_q$ satisfy

$$\theta_p(s) < \theta_q(s)$$

at some sample point $s$. If their ordering is reversed elsewhere on a connected region, continuity implies that they must coincide at an intermediate point. Such an intersection is witnessed algebraically by a zero of the resultant of the defining polynomials. This relationship is central to the use of resultants in both classical CAD and levelwise single-cell construction [McC98, Bro01, NÁS+24].

**Definition 2.1.6** (Discriminant [Nal25]). *Let $p(x)$ be a polynomial of degree $m > 0$ over a field $K$. The* discriminant *of $p$ with respect to $x$, denoted by*

$$\mathrm{disc}_x(p),$$

*is, up to a factor depending on the degree and leading coefficient, the resultant of $p$ and its derivative. In particular,*

$$\mathrm{disc}_x(p) = 0$$

*if and only if $p$ has a multiple root over an algebraic closure of $K$.*

As with resultants, the discriminant of a multivariate polynomial can be computed with respect to its highest variable:

$$\mathrm{disc}_{x_i}(p) \in \mathbb{Q}[x_1,\ldots,x_{i-1}].$$

Its zeros identify points at which roots of $p$ may collide or change multiplicity.

The leading coefficient

$$\mathrm{ldcf}_{x_i}(p)$$

also plays an important role. If it vanishes at a point $s$ and $p(s,x_i)$ is not the zero polynomial, then the degree of $p(s,x_i)$ with respect to $x_i$ is strictly smaller than $\deg_{x_i}(p)$. If all coefficients of $p$ vanish at $s$, then $p(s,x_i) = 0$, that is, $p$ is nullified at $s$ in the sense of Definition 2.1.2. Thus, zeros of the discriminant indicate possible multiple roots, whereas zeros of the leading coefficient indicate a drop in degree, and simultaneous zeros of all coefficients characterize nullification.

**Example 2.1.3.** *Consider the polynomials*

$$p = x_2 - x_1 \qquad \textit{and} \qquad q = x_2 + x_1 - 2.$$

*Both are of level* 2*. Their roots with respect to* $x_2$ *are*

$$\theta_p(x_1) = x_1, \qquad \theta_q(x_1) = 2 - x_1.$$

*The root functions intersect if*

$$x_1 = 2 - x_1,$$

*that is, at* $x_1 = 1$*. Correspondingly,*

$$\mathrm{res}_{x_2}(p,q)$$

*has a root at* $x_1 = 1$*.*

*Now consider*

$$r = x_2^2 - x_1.$$

*With respect to* $x_2$*, its discriminant is proportional to*

$$x_1.$$

*At* $x_1 = 0$*, the two roots*

$$-\sqrt{x_1} \qquad \textit{and} \qquad \sqrt{x_1}$$

*meet and form a multiple root. Hence, the discriminant detects the point at which the structure of the real roots changes.*

*Resultants and discriminants therefore detect two different kinds of critical events: resultants detect intersections between roots of different polynomials, whereas discriminants detect collisions between roots of the same polynomial.*

### 2.1.4 Sign and Order Invariance

The purpose of CAD is not merely to isolate polynomial roots at individual points, but to construct regions over which relevant polynomial properties remain invariant.

**Definition 2.1.7** (Sign Invariance [Nal25])**.** *Let*

$$p \in \mathbb{Q}[x_1, \ldots, x_i]$$

*and let* $R \subseteq \mathbb{R}^i$*. The polynomial* $p$ *is* sign-invariant *on* $R$ *if*

$$\mathrm{sign}(p(r)) = \mathrm{sign}(p(r'))$$

*for all* $r,r' \in R$*.*

Equivalently, exactly one of

$$p < 0, \qquad p = 0, \qquad p > 0$$

holds throughout $R$.

Sign invariance allows the truth of a polynomial constraint to be determined by evaluating the corresponding polynomial at a single sample point. If $s \in R$ and $p$ is sign-invariant on $R$, then the sign of $p(s)$ determines the sign of $p$ throughout the entire region.

To formulate the stronger invariance property required by several CAD projection theorems, we first define the order of a polynomial at a point.

**Definition 2.1.8** (Order of a Polynomial [Nal25])**.** *Let $p \in \mathbb{Q}[x_1, \ldots, x_i]$ and let $r \in \mathbb{R}^i$. The* order *of $p$ at $r$, denoted by $\mathrm{ord}_r(p)$, is the smallest $k \in \mathbb{N}$ for which a partial derivative of $p$ of total order $k$ does not vanish at $r$.*

*If every partial derivative of $p$ vanishes at $r$, we set*

$$\mathrm{ord}_r(p) = \infty.$$

In particular, $\mathrm{ord}_r(p) = 0$ whenever $p(r) \neq 0$. For a univariate polynomial and a root $r$, the order is exactly the multiplicity of that root.

**Definition 2.1.9** (Order Invariance [Nal25])**.** *Let*

$$p \in \mathbb{Q}[x_1, \ldots, x_i]$$

*and $R \subseteq \mathbb{R}^i$. The polynomial $p$ is* order-invariant *on $R$ if*

$$\mathrm{ord}_r(p) = \mathrm{ord}_{r'}(p)$$

*for all $r$,$r' \in R$.*

Order invariance controls not only whether a polynomial vanishes, but also how it vanishes. This is particularly relevant on sections of polynomial varieties, where projection theorems require the multiplicity of the corresponding root to remain constant [McC98, Bro01].

**Example 2.1.4.** *Consider*

$$p(x) = x^2.$$

*On the interval $R = (-1,0)$, the polynomial is positive and therefore sign-invariant. Since it never vanishes there, it is also order-invariant with*

$$\mathrm{ord}_r(p) = 0$$

*for every $r \in R$.*

*Now consider the set*

$$R' = \{0,1\}$$

*and*

$$q(x) = x^2(x-1).$$

*The polynomial vanishes at both points and is therefore sign-invariant on $R'$. However,*

$$\mathrm{ord}_0(q) = 2 \qquad \textit{and} \qquad \mathrm{ord}_1(q) = 1.$$

*Hence, $q$ is not order-invariant on $R'$.*

*Sign invariance alone therefore does not capture the multiplicity information required by CAD projection theory.*

### 2.1.5 Delineability

The notions introduced so far describe individual algebraic events. Delineability combines them into a regularity property describing how all roots of a polynomial behave over a lower-dimensional region.

**Definition 2.1.10** (Delineability [Nal25])**.** *Let*

$$p \in \mathbb{Q}[x_1, \ldots, x_{i+1}] \qquad \text{and} \qquad R \subseteq \mathbb{R}^i$$

*with $R$ connected. The polynomial $p$ is* delineable *on $R$ if its real roots over $R$ can be described by continuous root functions*

$$\theta_1 < \cdots < \theta_k$$

*such that, at every $r \in R$, the values $\theta_1(r), \ldots, \theta_k(r)$ are exactly the distinct real roots of $p(r, x_{i+1})$, and the multiplicity of each of these roots remains constant throughout $R$.*

In CAD, delineability provides the root structure required for lifting [McC98, Bro01]. If $p$ is delineable over a cell, its roots cannot unexpectedly appear, disappear, or cross one another within that cell. Consequently, the root functions provide a consistent decomposition of the cylinder over the cell.

**Definition 2.1.11** (Analytic Delineability [Nal25])**.** *Let $R \subseteq \mathbb{R}^i$ be a connected analytic submanifold. A polynomial $p \in \mathbb{Q}[x_1, \ldots, x_{i+1}]$ is* analytically delineable *on $R$ if it is delineable on $R$ and its real root functions can be chosen to be analytic.*

Here, a function is *analytic* if it can locally be represented by a convergent power series. An *analytic submanifold* is a submanifold that locally admits analytic coordinate descriptions, with analyticity of functions on it understood in these coordinates [Nal25]. The corresponding cell property of the proof system additionally records the order invariance of the polynomial on its sections; this property is introduced formally together with the proof system in Section 2.3.

**Example 2.1.5.** *Consider again*

$$p(x_1, x_2) = x_2^2 - x_1.$$

*On*

$$R = (0, \infty),$$

*the two real roots are described by*

$$\theta_1(x_1) = -\sqrt{x_1}, \qquad \theta_2(x_1) = \sqrt{x_1}.$$

*Both functions are analytic on $R$, they remain strictly ordered, and both roots have constant multiplicity one. Hence, $p$ is analytically delineable on $R$.*

*The same is not true on an interval containing $x_1 = 0$. At $x_1 = 0$, the two roots coincide, while for $x_1 < 0$ there are no real roots. Thus, neither the number nor the multiplicity structure of the real roots remains constant.*

*Correspondingly,*

$$\operatorname{disc}_{x_2}(p) = 4x_1$$

*vanishes at the critical point $x_1 = 0$. Restricting the base cell so that this discriminant is sign- or order-invariant prevents the lifting procedure from crossing this change in root structure.*

## 2.2 Cylindrical Algebraic Decomposition

*Cylindrical algebraic decomposition* (CAD) is a method for decomposing real space into finitely many regions on which a given set of polynomials exhibits invariant behaviour [Col75]. Originally introduced as a method for quantifier elimination over the real numbers, CAD has since become an important technique for reasoning about systems of polynomial constraints.

The central idea is to partition $\mathbb{R}^n$ into cells such that the signs of the relevant polynomials do not change within a cell. Consequently, the truth of a Boolean combination of polynomial constraints can be determined by evaluating the constraints at a single representative *sample point* of each cell.

### 2.2.1 Cylindrical Decompositions

We first introduce the notion of a cell.

**Definition 2.2.1** (Cell [NÁS+24])**.** *A* cell *in* $\mathbb{R}$ *is either a point* $\{a\}$ *or an interval* $(a,b)$*, where the bounds may also be* $-\infty$ *or* $+\infty$*.*

*Let* $R \subseteq \mathbb{R}^{i-1}$ *be a cell and let*

$$f,g : R \to \mathbb{R}$$

*be continuous functions with* $f(r) < g(r)$ *for all* $r \in R$*. A cell in* $\mathbb{R}^i$ *above* $R$ *is either a* section

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

*or a* sector*, for example*

$$\{(r,x_i) \mid r \in R,\, f(r) < x_i < g(r)\}.$$

*The analogous unbounded sectors below* $f$ *or above* $g$ *are defined accordingly.*

A decomposition is cylindrical over a lower-dimensional decomposition if the projection of every cell is itself a cell of that decomposition [NÁS+24].

**Definition 2.2.2** (Cylindrical Decomposition)**.** *A finite partition* $D$ *of* $\mathbb{R}^n$ *into cells is called a* cylindrical decomposition *if, for any two cells* $R,R' \in D$ *and every* $i < n$*, their projections onto the first* $i$ *coordinates are either identical or disjoint.*

Informally, cells are arranged in *stacks* above cells of lower dimension. In two dimensions, for example, the cells above an interval on the $x_1$-axis consist of sections given by root functions and sectors between these sections. Two such stacks cannot partially overlap in their projection onto the $x_1$-axis.

For reasoning about polynomial constraints, cylindricity alone is not sufficient. The decomposition must additionally respect the polynomials occurring in the problem. The term algebraic refers to the requirement that the cells are *semialgebraic*, meaning that they can be described by finite Boolean combinations of polynomial equations and inequalities.

**Definition 2.2.3** (Sign-Invariant CAD [NÁS+24])**.** *Let* $P \subseteq \mathbb{Q}[x_1,\ldots,x_n]$ *be a finite set of polynomials. A* sign-invariant cylindrical algebraic decomposition *for* $P$ *is a cylindrical decomposition of* $\mathbb{R}^n$ *into semialgebraic cells such that every polynomial* $p \in P$ *is sign-invariant on every cell of the decomposition.*

If a cell $R$ is sign-invariant for all polynomials appearing in a quantifier-free formula, then the truth value of the formula is constant throughout $R$. It therefore suffices to select one sample point $s \in R$ and evaluate the formula at $s$.

### 2.2.2 Projection and Lifting

A CAD is constructed in two phases: *projection* and *lifting* [Col75, McC98]. Suppose that the input consists of a finite set

$$P_n \subseteq \mathbb{Q}[x_1, \ldots, x_n].$$

During projection, information about the roots of polynomials at level $i$ is encoded by polynomials of lower level. This produces a sequence

$$P_n, P_{n-1}, \ldots, P_1,$$

where

$$P_i \subseteq \mathbb{Q}[x_1, \ldots, x_i].$$

*Projection polynomials* typically include discriminants, coefficients, and resultants [McC98, Bro01]. As discussed in Section 2.1.3, these polynomials identify locations at which the structure or ordering of higher-dimensional roots may change. By ensuring suitable invariance properties for the projection polynomials, the roots of the original polynomials can be shown to behave regularly over the resulting lower-dimensional cells.

Once the projection has reached the univariate level, the construction proceeds in the opposite direction through lifting. The real roots of the polynomials in $P_1$ partition the real line into point cells and open intervals. A sample point is chosen for each of these cells.

Given a cell

$$R \subseteq \mathbb{R}^{i-1}$$

and a sample point

$$s \in R,$$

the polynomials in $P_i$ are specialized at $s$. Their real roots are isolated and ordered,

$$\rho_1 < \rho_2 < \cdots < \rho_k,$$

and partition the fibre above $s$ into sections and sectors:

$$(-\infty, \rho_1),\ \{\rho_1\},\ (\rho_1, \rho_2),\ \ldots,\ \{\rho_k\},\ (\rho_k, \infty).$$

If the projection conditions guarantee delineability over $R$, these roots extend to root functions

$$\theta_1, \ldots, \theta_k : R \to \mathbb{R}.$$

The corresponding sections and sectors then form a stack of cells over $R$. Repeating this procedure until level $n$ yields a cylindrical decomposition of $\mathbb{R}^n$.

**Example 2.2.1.** *Consider the polynomials* $p_1 = x_2^2 - x_1$ *and* $p_2 = x_2 - 1$ *with the variable ordering* $x_1 \prec x_2$.

*The polynomial* $p_1$ *has a multiple root with respect to* $x_2$ *at* $x_1 = 0$*, while* $p_1$ *and* $p_2$ *share a root at* $x_1 = 1$*. These critical values are detected by*

$$\mathrm{disc}_{x_2}(p_1) \quad \textit{and} \quad \mathrm{res}_{x_2}(p_1,p_2),$$

*respectively.*

*The projection therefore partitions the* $x_1$*-axis at the critical points* 0 *and* 1*, resulting in cells such as*

$$(-\infty,0), \quad \{0\}, \quad (0,1), \quad \{1\}, \quad (1,\infty).$$

*Consider the cell*

$$R = (0,1).$$

*Above this interval, the roots of* $p_1$ *are described by*

$$-\sqrt{x_1} \qquad \textit{and} \qquad \sqrt{x_1},$$

*while the root of* $p_2$ *is the constant function* 1*. Throughout* $R$*, these functions remain ordered as*

$$-\sqrt{x_1} < \sqrt{x_1} < 1.$$

*They consequently divide the cylinder* $R \times \mathbb{R}$ *into sections and sectors on which both input polynomials are sign-invariant.*

*Repeating the lifting procedure for every cell on the* $x_1$*-axis produces a sign-invariant CAD of* $\mathbb{R}^2$*.*

### 2.2.3 From Full CAD to Local Cell Construction

The construction described above computes cells covering the entire space. A complete sign-invariant CAD provides a general method, for example for quantifier elimination, but it is not inherently required: depending on the formula and the method, part of a decomposition may suffice. Many modern applications, in particular satisfiability checking, require information only about selected regions of $\mathbb{R}^n$ [ÁDEK21, NÁS+24].

Suppose, for example, that during a satisfiability search a particular sample point

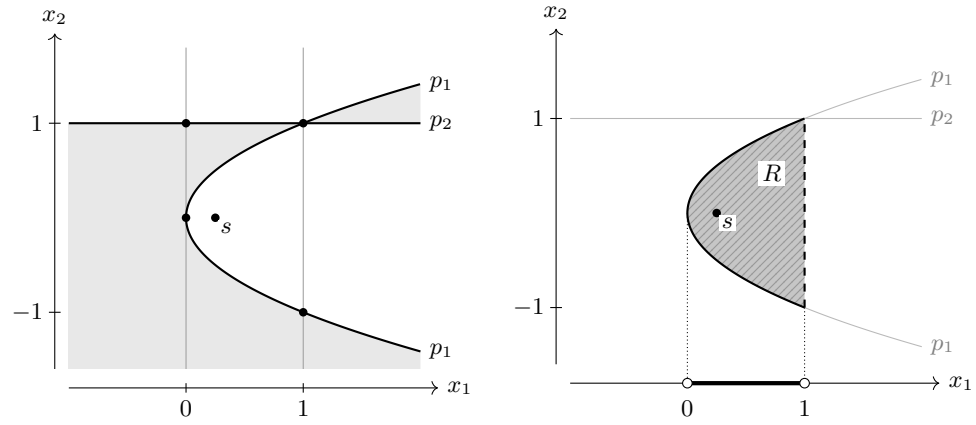
$$s \in \mathbb{R}^n$$

has already been found to violate a constraint. Rather than computing a complete CAD, it can be sufficient to construct a single cell

$$R \subseteq \mathbb{R}^n$$

containing $s$ on which the relevant polynomial signs remain invariant. Every point in this cell then violates the constraint for the same reason, allowing the entire region to be excluded from further search. Figure 2.1 contrasts both approaches for the polynomials of Example 2.2.1.

*Exploration-guided CAD* algorithms exploit this observation by tailoring projection to a given partial sample point [ÁDEK21, NÁS+24]. In contrast to a full CAD projection, not every potential interaction between polynomial roots must necessarily be preserved. Only those conditions required to justify the cell currently being constructed are relevant. This local perspective can avoid projection polynomials that would be necessary for a complete decomposition. Section 2.3 describes one such method, the levelwise single-cell construction.




(a) Full CAD (schematic)

(b) Levelwise single-cell construction

Figure 2.1: Schematic comparison of a complete sign-invariant CAD and levelwise single-cell construction for $p_1 = x_2^2 - x_1$ and $p_2 = x_2 - 1$. (a) The complete CAD partitions $\mathbb{R}^2$ into 27 cells, arranged in stacks over $(-\infty,0)$, $\{0\}$, $(0,1)$, $\{1\}$, and $(1,\infty)$. (b) For $s = (1/4,0)$, single-cell construction determines only $R = \{(x_1,x_2) \mid 0 < x_1 < 1,\ -\sqrt{x_1} < x_2 < \sqrt{x_1}\}$. This is the same cell that contains $s$ in (a), but the other cells of the complete CAD need not be constructed.

## 2.3 Levelwise Single-Cell Construction

The *levelwise single-cell construction* realizes the local approach of Section 2.2.3. Given a finite set of polynomials and a sample point, the algorithm constructs one cylindrical cell around the sample by processing the variables one level at a time [NÁS+24]. At each level, the roots of the currently relevant polynomials are considered at the corresponding *partial sample point*. These roots determine a symbolic description of the cell in the current dimension. Projection then determines which lower-level polynomial properties are sufficient to guarantee that this symbolic description remains valid over an entire lower-dimensional cell.

Since only the root relationships required for the cell containing the current sample are maintained, different choices during the construction can lead to different valid cells and different sets of projection polynomials.

### 2.3.1 Indexed Root Expressions

A single-cell construction must refer symbolically to the roots that form the boundaries of the cell. For this purpose, indexed root expressions are used.

**Definition 2.3.1** (Indexed Root Expression [NÁS+24])**.** *Let $p \in \mathbb{Q}[x_1,\dots,x_i]$ be a polynomial of level $i$, and let $k \in \mathbb{N}_{>0}$. For a partial sample point $s \in \mathbb{R}^{i-1}$, the* indexed root expression

$$\mathrm{root}_{x_i,k}(p)$$

*is defined at $s$ if $p(s,x_i)$ is not the zero polynomial and has at least $k$ distinct real roots. If these roots are ordered as*

$$r_1 < \cdots < r_m,$$

*then*

$$\mathrm{root}_{x_i,k}(p)\,(s) = r_k.$$

*Otherwise, the indexed root expression is undefined.*

If the meaning is clear from the context, we write an indexed root expression simply as $\xi$. We denote its *defining polynomial* by $\xi.p$ and its *root index* by $\xi.k$.

Indexed root expressions differ from the real root functions introduced in Definition 2.1.4. An indexed root expression selects a root according to its position among the real roots at a particular point. It is not necessarily defined continuously throughout all of $\mathbb{R}^{i-1}$. For example, roots may merge at a discriminant root or disappear when the degree of the defining polynomial changes.

An indexed root expression selects the $k$-th distinct real root at each partial sample point where it is defined. This selection may be discontinuous, whereas a real root function is continuous on its domain. This connection between indexed root expressions and root functions is later captured explicitly by the properties of the CAD proof system.

### 2.3.2 Symbolic Intervals

At a fixed sample point, the real roots of the relevant polynomials partition the current coordinate axis into sections and sectors. Instead of storing only the numerical endpoints of the interval containing the sample, the levelwise construction uses indexed root expressions as symbolic bounds.

**Definition 2.3.2** (Symbolic Interval [NÁS+24])**.** *A* symbolic interval *of level $i$ is either a section*

$$[\xi,\xi],$$

*determined by an indexed root expression $\xi$ of level $i$, or a sector*

$$(l,u),$$

*where each finite bound is an indexed root expression of level $i$ and either bound may instead be $-\infty$ or $+\infty$, respectively.*

At a partial sample point where its finite bounds are defined, a symbolic interval represents the corresponding point or open interval on the $x_i$-axis. Whether this interval contains the current sample is imposed later as part of the representation.

Suppose, for instance, that the relevant specialized polynomials at

$$s_{[i-1]} = (s_1, \ldots, s_{i-1})$$

have roots

$$\xi_1(s_{[i-1]}) < \xi_2(s_{[i-1]}) < \xi_3(s_{[i-1]}),$$

and

$$\xi_1(s_{[i-1]}) < s_i < \xi_2(s_{[i-1]}).$$

The current cell can then be described symbolically by

$$I = (\xi_1,\xi_2).$$

The symbolic interval describes more than the concrete interval

$$\big(\xi_1(s_{[i-1]}),\xi_2(s_{[i-1]})\big)$$

at the sample point. Its indexed-root bounds must remain defined and correctly ordered while the lower coordinates range over the entire underlying cell.

### 2.3.3 Indexed Root Orderings

Choosing the two boundaries of a symbolic interval is not sufficient to establish sign invariance. Roots that lie outside the interval at the current sample must not cross its boundaries as the lower coordinates vary. The single-cell construction records the necessary relationships using an indexed root ordering.

**Definition 2.3.3** (Indexed Root Ordering [NÁS+24])**.** *Let $\Xi$ be a finite set of indexed root expressions of level $i$. An* indexed root ordering $\preceq$ *on $\Xi$ records selected ordering relations between elements of $\Xi$, with the induced relations forming a consistent partial ordering.*

*Such an ordering* matches *a partial sample point $s \in \mathbb{R}^{i-1}$ if every indexed root expression occurring in the ordering is defined at $s$ and every recorded comparison holds for the corresponding root values at $s$.*

The ordering need not explicitly compare every pair of roots. Missing comparisons may follow by transitivity, which allows different valid orderings to encode the relationships needed for the same cell. Thus, the ordering is generally not unique. This freedom influences which resultants are required during projection, since maintaining an ordering between roots of different polynomials typically requires reasoning about the resultant of their defining polynomials [NÁS+24].

**Example 2.3.1.** *Let $\Xi = \{\xi_1,\xi_2,\xi_3,\xi_4\}$ be indexed root expressions satisfying $\xi_1(s) < s_i < \xi_2(s) < \xi_3(s) < \xi_4(s)$ at the current partial sample point.*

*Assume that*

$$I = (\xi_1,\xi_2)$$

*is chosen as the symbolic interval. In order to prevent $\xi_3$ and $\xi_4$ from entering $I$, one may choose*

$$\xi_2 \prec \xi_3, \qquad \xi_2 \prec \xi_4.$$

*Alternatively, one may use*

$$\xi_2 \prec \xi_3 \prec \xi_4.$$

*If all roots are defined by different polynomials, the first choice requires maintaining relationships corresponding to the resultants between the defining polynomial of $\xi_2$ and the defining polynomials of both $\xi_3$ and $\xi_4$. The second choice instead uses a resultant between the polynomials defining $\xi_2$ and $\xi_3$, and one between those defining $\xi_3$ and $\xi_4$.*

*Both choices can therefore describe a correct cell while generating different projection polynomials. This non-uniqueness is one of the motivations for separating the representation choice from its correctness proof.*

### 2.3.4 Representations

At each level, the construction fixes two central pieces of information about the cell. The symbolic interval specifies the intended boundaries of the cell in the $x_i$-direction, and the indexed root ordering records which relationships between polynomial roots must be preserved over the underlying cell. Together, they form a representation of the cell at this level. In the section case, polynomial equations that hold throughout the section can be exploited to reduce the required projection conditions. The representation records the corresponding polynomials in a set $E$, which must be empty in the sector case [NÁS+24].

**Definition 2.3.4** (Representation for $\Xi$ [NÁS$^+$24])**.** *Let $\Xi$ be a set of indexed root expressions of level $i$ and let $s$ be the current sample point. A* representation for $\Xi$ at $s$ *is a tuple*

$$(I,E,\preceq),$$

*consisting of a symbolic interval $I$ of level $i$, a set $E$ of level-$i$ polynomials used as* equational constraints*, and an indexed root ordering $\preceq$ on $\Xi$.*

*The tuple is a* valid *representation if $I$ contains the current sample coordinate $s_i$, $\preceq$ matches the corresponding partial sample point $s_{[i-1]}$, and every expression in $\Xi$ is accounted for either by the ordering or by placing its defining polynomial in $E$. If $I$ is a sector, then $E$ must be empty.*

The representation contains the information needed to determine which projection conditions must be imposed at the next lower level. Importantly, representations are generally not unique. The algorithm may choose different symbolic boundaries, different root orderings, or, in the section case, exploit different equational constraints.

### 2.3.5 Levelwise Construction

Let

$$P \subseteq \mathbb{Q}[x_1,\dots,x_n]$$

be a finite set of input polynomials and

$$s = (s_1,\dots,s_n) \in \mathbb{R}^n$$

a sample point. The goal of single-cell construction is to construct a cylindrical cell

$$R \subseteq \mathbb{R}^n$$

such that

$$s \in R$$

and all relevant input polynomials are sign-invariant on $R$.

The construction proceeds from the highest level towards the lowest level. At level $i$, the algorithm considers the polynomials that currently need to be invariant and specializes them at

$$s_{[i-1]} = (s_1,\dots,s_{i-1}).$$

The real roots of these polynomials are isolated and ordered. From these roots, a representation

$$(I_i,E_i,\preceq_i)$$

is selected.

The representation determines the cell in the $x_i$-direction and the relationships between root functions that must be preserved. Projection then generates a set of lower-level polynomials whose invariance is sufficient to preserve these relationships. The procedure is subsequently repeated at level $i-1$.

Schematically, the computation has the form

$$P_n \xrightarrow{(I_n,E_n,\preceq_n)} P_{n-1} \xrightarrow{(I_{n-1},E_{n-1},\preceq_{n-1})} \cdots \xrightarrow{(I_2,E_2,\preceq_2)} P_1.$$

At the univariate level, the required properties can be established directly by root isolation. The symbolic intervals

$$I_1, \ldots, I_n$$

then describe the resulting cylindrical cell.

Unlike a full CAD projection, the set of lower-dimensional projection polynomials depends on the selected representation. A resultant is only needed if the chosen ordering requires the corresponding roots to remain ordered [NÁS+24].

Several representations may satisfy the requirements, but they can produce cells of different sizes and projection problems of different complexity. The algorithm therefore has to choose between them using heuristics. The proof system introduced next ensures that correctness does not depend on which heuristic is used.

### 2.3.6 Proof System for Single-Cell Construction

The projection procedure described above intertwines two conceptually different tasks. First, the algorithm must decide *how* to describe the cell, for example by selecting symbolic bounds and a root ordering. Second, it must justify that these choices are sufficient to establish the required invariance properties.

The CAD proof system separates these concerns [NÁS+24]. Instead of directly transforming one set of projection polynomials into another, it reasons about *properties* of the cell being constructed. Inference rules reduce a desired property on the current level to a collection of simpler properties, usually on the underlying cell. The algorithm then acts as a proof strategy that selects which rules to apply and how to instantiate them.

For the formalization in this thesis, this means that the heuristic choice of a representation does not itself have to be verified. Instead, the generated sequence of inference-rule applications can be checked independently.

#### Properties

A property describes a statement about the cell currently under construction. The level specifies the dimension of the geometric object on which the property is evaluated.

For example, the property

$$\text{sgn_inv}(p)$$

for a polynomial $p$ of level $i$ states that $p$ is sign-invariant on the current cell. The property

$$\text{sample}(s)$$

states that the sample point $s$ is contained in the cell. Other properties encode the geometric structure needed to reason about root functions.

The basic proof system uses several recurring properties. For the purposes of this thesis, the most relevant ones are summarized below.

**Definition 2.3.5** (Basic Cell Properties [NÁS+24])**.** *Let $R \subseteq \mathbb{R}^i$, let $s \in \mathbb{R}^i$, let $p \in \mathbb{Q}[x_1, \ldots, x_i]$ be a polynomial of level $i$, and let $q \in \mathbb{Q}[x_1, \ldots, x_{i+1}]$ be a polynomial*

*of level $i+1$. The basic proof system uses the following properties:*

$$\begin{aligned}
\text{sample}(s)(R) &\iff s \in R,\\
\text{sgn_inv}(p)(R) &\iff p \text{ is sign-invariant on } R,\\
\text{ord_inv}(p)(R) &\iff p \text{ is order-invariant on } R,\\
\text{non_null}(q)(R) &\iff q \text{ is not nullified at any point of } R,\\
\text{an_sub}(i)(R) &\iff R \text{ is an analytic submanifold},\\
\text{connected}(i)(R) &\iff R \text{ is connected.}
\end{aligned}$$

*Moreover,* $\text{an_del}(q)(R)$ *states that $R$ is a connected analytic submanifold, that $q$ is analytically delineable over $R$, and that $q$ is order-invariant on its sections.*

These predicates allow the inference rules to refer uniformly to algebraic and geometric requirements on the cell being constructed.

The proof system additionally contains properties associated with the symbolic representation of a cell.

**Definition 2.3.6** (Representation Property [NÁS+24])**.** *Let $I$ be a symbolic interval of level $i$, let $s \in \mathbb{R}^{i-1}$, and let $R \subseteq \mathbb{R}^i$.*

*The property*

$$\text{repr}(I,s)(R)$$

*holds if the indexed root expressions forming the bounds of $I$ refer to well-defined real root functions over the projection*

$$R_{\downarrow[i-1]},$$

*and these root functions describe exactly the lower and upper boundaries of $R$ in the $x_i$-direction.*

Thus, $\text{repr}(I,s)$ connects the syntactic object $I$ used by the algorithm to the geometric cell that it is intended to describe.

Likewise, the ordering selected by the algorithm is represented by a cell property.

**Definition 2.3.7** (Indexed Root Ordering Property [NÁS+24])**.** *Let $\preceq$ be an indexed root ordering of level $i$, let $s \in \mathbb{R}^{i-1}$, and let $\preceq^t$ denote its reflexive and transitive closure.*

*The property*

$$\text{ir_ord}(\preceq,s)$$

*holds on a set $R \subseteq \mathbb{R}^{i-1}$ if $\preceq$ matches $s$, every indexed root expression $\xi$ occurring in the ordering determines a real root function $\theta_{\xi,s}$ on $R$, and the ordering induced at $s$ is preserved throughout $R$.*

*More precisely, if $\xi \preceq^t \xi'$, then for every $r \in R$,*

$$\xi(s) < \xi'(s) \quad \Longrightarrow \quad \theta_{\xi,s}(r) < \theta_{\xi',s}(r),$$

*while*

$$\xi(s) = \xi'(s) \quad \Longrightarrow \quad \theta_{\xi,s}(r) = \theta_{\xi',s}(r).$$

At the sample point, it is sufficient to compare algebraic root values. The property ir_ord, however, states the stronger geometric fact that the same relationship persists over the complete underlying cell.

Finally, the proof system distinguishes between the symbolic description of a cell and the assumption that this symbolic description is actually used to construct it.

**Definition 2.3.8** (Cell Description Property [Nal25, Property 4.4])**.** *Let $I$ be a symbolic interval of level $i$. The property* holds($I$) *holds on a cell $R \subseteq \mathbb{R}^i$ if $R$ is obtained by extending its projection $R_{\downarrow[i-1]}$ according to the symbolic interval $I$, that is, if $R$ consists exactly of the points above $R_{\downarrow[i-1]}$ that lie in the section or sector described by $I$.*

The property holds($I$) is special within the proof system: it is treated as an assumption describing the cell being constructed rather than as a property that must itself be reduced by another proof rule. The proof system [NÁS$^+$24] explicitly identifies 'holds(I)' as the endpoint of this reduction.

### Inference Rules

An inference rule describes how a desired cell property can be derived from other properties.

**Definition 2.3.9** (Inference Rule [NÁS$^+$24])**.** *An* inference rule *of the CAD proof system has the form*

$$P_1, \ldots, P_k \vdash Q,$$

*where the* antecedents $P_1, \ldots, P_k$ *and the* consequent $Q$ *are cell properties. The rule is* sound *if, whenever all antecedents hold on a cell $R$, the consequent holds on $R$ as well:*

$$P_1(R) \wedge \cdots \wedge P_k(R) \implies Q(R).$$

Rather than presenting every rule of the proof system, we illustrate its operation using the rules most relevant to the formalization in this thesis.

**Establishing delineability.** A central rule reduces analytic delineability of an irreducible polynomial to lower-dimensional conditions [Nal25, Rule 4.1]:

$$\begin{aligned} &\text{an_sub}(i)(R),\ \text{connected}(i)(R),\ \text{non_null}(p)(R),\\ &\text{ord_inv}(\text{disc}_{x_{i+1}}(p))(R),\ \text{sgn_inv}(\text{ldcf}_{x_{i+1}}(p))(R)\\ &\qquad\qquad\qquad\qquad \vdash \text{an_del}(p)(R). \end{aligned}$$

The rule captures a familiar CAD projection principle [McC98, Bro01]. If the discriminant does not change its order, the leading coefficient does not change its sign, and the polynomial is never nullified, then its roots behave regularly over the underlying cell. The geometric statement of delineability is therefore reduced to algebraic properties of lower-level projection polynomials.

The rules of the proof system generally follow this pattern:

high-dimensional geometric property $\quad\rightarrow\quad$ lower-dimensional algebraic obligations.

**Maintaining a root ordering.** Suppose the selected representation requires a root function of $p$ to remain below a root function of $q$. At the current sample point this relation can be checked directly. To guarantee that the order remains unchanged throughout the underlying cell, the proof system uses delineability together with the resultant of the two defining polynomials.

In simplified form, the corresponding rule [Nal25, Rule 4.9] establishes

$$\text{ir_ord}(\preceq, s)(R)$$

from

$$\text{sample}(s)(R), \qquad \text{an_sub}(i)(R), \qquad \text{connected}(i)(R),$$

the delineability of the defining polynomials, and the order invariance of the relevant resultants.

The underlying argument is that two continuous root functions cannot change their strict ordering without intersecting. Such an intersection would be witnessed by a zero of their resultant. The proof-system rule formalizes precisely this relationship. In the full system, the rule reasons over a partial ordering and uses transitivity, so a resultant need only be introduced for the explicitly selected ordering relations [NÁS+24].

**Deriving sign invariance.** Once the cell is represented by a symbolic interval and all roots of an irreducible polynomial are guaranteed to remain outside that interval, sign invariance can be concluded.

Schematically, Rule 4.10 [Nal25, Rule 4.10] has the form

$$\text{sample}(s)(R),\ \text{repr}(I,s)(R),\ \text{ir_ord}(\preceq, s)(R),\ \text{an_del}(p)(R) \vdash \text{sgn_inv}(p)(R).$$

The rule additionally requires that $p$ has at least one real root at $s$, that the defining polynomials of the indexed root expressions in $\preceq$ are irreducible, and that $\preceq$ matches $s$. Furthermore, the ordering must establish that every root of $p$ lies on or outside the corresponding cell boundaries, possibly using transitivity.

This rule explains the role of the representation. The symbolic interval specifies the intended cell, while the root ordering ensures that no root can enter its interior. Delineability guarantees that no additional roots appear unexpectedly. Together, these facts establish that $p$ cannot change sign inside the cell [NÁS+24].

If a polynomial has no real root above the current sample, an even simpler rule applies: delineability alone guarantees that it has no real root anywhere above the connected underlying cell and is therefore sign-invariant there. The proof system makes this reduction explicit.

**Establishing the cell representation.** The property $\text{repr}(I,s)$ itself must also be justified. Assume first that every finite bound of $I$ denotes a valid indexed root at the sample $s$. If the cell is constructed according to $I$, expressed by $\text{holds}(I)$, and the polynomials defining its finite boundaries are delineable, then these bounds correspond to well-defined real root functions over the underlying cell.

For a sector $I = (l,u)$, Rule 4.13 [Nal25, Rule 4.13] has the form

$$\text{sample}(s)(R),\ \text{holds}(I)(R),\ (l = -\infty \vee \text{an_del}(l.p)(R)),\ (u = \infty \vee \text{an_del}(u.p)(R)) \vdash \text{repr}(I,s)(R).$$

This rule is conceptually important for the formalization: it separates the fact that the algorithm has *chosen* a symbolic interval from the mathematical statement that its symbolic bounds really describe continuous cell boundaries. The source derives ‘repr‘ from ‘holds‘ precisely through delineability of the boundary polynomials [NÁS+24].

**Connectedness and non-emptiness.** Some higher-level rules require the underlying cell to be connected. A symbolic interval does not automatically guarantee this. In particular, if its lower and upper root functions intersect, the set described by the interval may become empty or disconnected over parts of the base cell.

The proof system therefore contains explicit properties for connectedness and non-emptiness [Nal25, Rules 4.14–4.15]. Informally, if

$$I = (l,u)$$

and the root ordering guarantees

$$l(r) < u(r)$$

throughout the underlying cell, then the interval remains non-empty. Together with connectedness of the underlying cell and a valid representation, this also yields connectedness of the lifted cell.

Making connectedness and non-emptiness explicit is useful for verification because an algorithmic description may otherwise rely on these conditions implicitly. In the proof system they become ordinary proof obligations and can therefore be checked separately.

### Coverings

The proof system is not restricted to individual cells. A *cylindrical algebraic covering* of $\mathbb{R}^n$ is a finite collection of semialgebraic cells whose union is $\mathbb{R}^n$. The cells are arranged cylindrically, meaning that their projections onto any lower-dimensional coordinate space are either identical or disjoint. Unlike in a cylindrical algebraic decomposition, the cells may overlap. For such coverings, the proof system must also express that several symbolic intervals jointly cover an entire cylinder over a lower-dimensional cell.

**Definition 2.3.10** (Covering Property [NK26])**.** *Let $C = (I_1, \dots, I_k)$ be a sequence of symbolic intervals of level $i+1$, and let $s \in \mathbb{R}^i$. The property* $\mathrm{cov}(C,s)$ *holds on $R \subseteq \mathbb{R}^i$ if the symbolic intervals $I_1, \dots, I_k$ can be represented over $R$ such that their corresponding cells cover the complete cylinder $R \times \mathbb{R}$.*

To establish the covering property, the proof system maintains appropriate orderings between the upper boundary of one interval and the lower boundary of the next. If

$$I_1, \dots, I_k$$

are ordered from bottom to top, the first interval must extend to $-\infty$, the final interval must extend to $+\infty$, and neighboring intervals must meet or overlap appropriately.

The individual cells are first characterized separately. Their lower-dimensional obligations are then combined, and the covering property is added to express that one of these cells contains every point in the cylinder.

### Applying Proof Rules

A proof-system-based projection step begins with a finite set $Q$ of properties that should hold on the cell. For a standard sign-invariant single-cell construction, this may initially be

$$Q = \{\mathrm{sgn_inv}(p) \mid p \in P\}.$$

The algorithm repeatedly selects a property

$$q \in Q$$

and an applicable inference rule

$$P_1, \ldots, P_k \vdash q.$$

The property $q$ is then replaced by its antecedents:

$$Q \leftarrow (Q \setminus \{q\}) \cup \{P_1, \ldots, P_k\}.$$

The proof rules are constructed such that antecedents are smaller than their consequents with respect to a suitable ordering on properties. Consequently, the core rule dependencies are acyclic; the source presents the resulting dependencies as a graph from high-level cell properties down toward representation assumptions and lower-level invariance properties.

A proof-system-based projection step can therefore be understood schematically as

$$\begin{aligned} \mathrm{sgn_inv}(p) &\Longleftarrow \mathrm{repr}(I,s), \mathrm{ir_ord}(\preceq, s), \mathrm{an_del}(p), \ldots \\ &\Longleftarrow \mathrm{holds}(I), \mathrm{ord_inv}(\mathrm{res}(\cdot,\cdot)), \mathrm{ord_inv}(\mathrm{disc}(\cdot)), \ldots \\ &\Longleftarrow \text{properties of lower level.} \end{aligned}$$

The projection algorithm based on the proof system takes a set $Q$, a sample point, a symbolic interval, and an indexed root ordering. It applies proof rules until all properties of the current level have been reduced. The remaining properties are returned as obligations for the next lower level.

**Example 2.3.2.** *Suppose that at level* 2 *we want to construct a cell around* $s = (s_1, s_2)$ *on which three irreducible polynomials* $p_1, p_2, p_3$ *are sign-invariant.*

*After specializing at* $s_1$*, assume that the relevant roots are*

$$\xi_1(s_1) < s_2 < \xi_2(s_1) < \xi_3(s_1) < \xi_4(s_1),$$

*and choose*

$$I_2 = (\xi_1, \xi_2).$$

*The initial proof obligations are*

$$\mathrm{sgn_inv}(p_1), \qquad \mathrm{sgn_inv}(p_2), \qquad \mathrm{sgn_inv}(p_3).$$

*For each polynomial, a sign-invariance rule reduces the obligation to properties including*

$$\mathrm{repr}(I_2, s_1), \qquad \mathrm{an_del}(p_j),$$

*and the root-ordering property needed to prevent the remaining roots from entering* $I_2$*.*

*A possible ordering is*

$$\xi_2 \prec \xi_3, \qquad \xi_2 \prec \xi_4.$$

*Maintaining these relations generates order-invariance obligations for the corresponding resultants.*

*In turn, each delineability obligation*

$$\mathrm{an_del}(p_j)$$

*is reduced to properties of its discriminant, leading coefficient, and non-nullification. After simplification, the remaining non-trivial obligations are polynomials of level* 1.

*Their roots are isolated at* $s_1$*, and an interval*

$$I_1$$

*containing* $s_1$ *is selected. Since these remaining polynomials are univariate, their required invariance can be established directly on* $I_1$*.*

*The final cell is then described levelwise by*

$$I_1 \quad \textit{and} \quad I_2,$$

*where the second-level bounds represented by* $\xi_1$ *and* $\xi_2$ *vary as root functions over* $I_1$*.*

*The algorithm thus constructs both a cell and a derivation establishing sign invariance of the input polynomials on that cell.*

## 2.4 Lean

Lean is an interactive theorem prover and functional programming language based on dependent type theory [dMU21]. It allows mathematical definitions, statements, and proofs to be represented in a formal language and mechanically checked for correctness. In contrast to conventional mathematical proofs, which leave many details implicit and rely on the reader to validate individual reasoning steps, a Lean proof must contain sufficient information for the Lean kernel to verify that the stated theorem follows from the underlying axioms and previously established results.

This thesis uses Lean to formalize parts of the proof system for cylindrical algebraic decomposition and to check concrete applications of its inference rules. Since Lean is used as the formal verification environment throughout the remainder of this thesis, this section introduces the concepts required to understand the formalization. The objective is not to provide a comprehensive introduction to Lean, but rather to explain its underlying proof model, the role of types and propositions, the use of tactics, and the distinction between proof construction and proof checking.

### 2.4.1 Types and Terms

At its core, Lean is based on a typed language. Every expression has a *type*, and the type determines which operations may be applied to it. For example, the declaration

```
def n : Nat := 3
```

defines a *term* `n` of type `Nat`. Similarly, a function from natural numbers to natural numbers can be written as

```
def double (n : Nat) : Nat :=
2 * n
```

The expression `double` itself has type

```
Nat -> Nat
```

and can therefore be applied to terms of type `Nat`.

Types are used throughout Lean not only to distinguish conventional data structures, but also to encode mathematical structure. Natural numbers, real numbers, polynomials, sets, functions, and algebraic structures are all represented by types and terms inhabiting them. Many mathematical relationships that are implicit on paper therefore become explicit typing constraints in Lean.

For example, when reasoning about objects in different dimensions, a paper presentation may use the same notation for points in $\mathbb{R}^n$ and $\mathbb{R}^{n-1}$, relying on the surrounding context to determine the intended dimension. In Lean, such distinctions can be encoded directly in the corresponding types. As a result, an operation that projects an $n$-dimensional point to $n-1$ dimensions must explicitly relate the two types involved.

### 2.4.2 Propositions as Types

The relationship between mathematical propositions and Lean's type system is given by the *Curry–Howard correspondence*. In Lean, propositions are themselves types, and a proof of a proposition is a term inhabiting the corresponding type [AdKU].

For example, the proposition

$$1 + 1 = 2$$

can be stated in Lean as

```
example : 1 + 1 = 2 := by
decide
```

Here, `1 + 1 = 2` is a term of type `Prop`. The proof following `:=` constructs a term witnessing that proposition.

This interpretation extends naturally to logical connectives. If $P$ and $Q$ are propositions, then an implication

$$P \to Q$$

is represented by a function type. A proof of $P \to Q$ is therefore a function that takes a proof of $P$ and produces a proof of $Q$. For example,

```
theorem identity_implication (P : Prop) : P -> P := by
intro h
exact h
```

constructs a proof of $P \to P$. The command `intro h` introduces the assumption $h : P$, and `exact h` uses this assumption as the proof of the conclusion.

This correspondence is particularly relevant for the formalization of the CAD proof system. A mathematical inference rule of the form

$$P_1, \ldots, P_k \vdash Q$$

can conceptually be represented in Lean by a theorem of the form

$$P_1 \to \cdots \to P_k \to Q.$$

Consequently, formally verifying an inference rule amounts to constructing a Lean term that transforms proofs of its antecedents into a proof of its consequent. Once

such a theorem has been established, a concrete application of the rule can be checked by providing proofs of the corresponding antecedents.

Conjunction and existential quantification are represented in a similar manner. A proof of $P \wedge Q$ contains both a proof of $P$ and a proof of $Q$, while a proof of an existential statement

$$\exists x,\, P(x)$$

contains a witness $x$ together with a proof of $P(x)$. This constructive interpretation of logic is central to how Lean represents mathematical proofs.

### 2.4.3 Dependent Types

A distinguishing feature of Lean is its support for *dependent types*. In an ordinary function type $A \to B$, the result type $B$ is independent of the argument. In a *dependent function type*, the result type itself may depend on the value of the argument [AdKU].

A simple example is the type `Fin n`. Its elements are natural numbers that are strictly smaller than $n$. The value $n$ therefore occurs as part of the type:

```
#check Fin 3
#check Fin 5
```

Although both types contain natural-number-like values, `Fin 3` and `Fin 5` are different types.

Dependent types make it possible to encode structural invariants directly in the type system. This is useful for mathematical formalizations where dimensions or indices form part of the semantics of an object. For example, an $n$-dimensional sample point and an indexed root expression defined over an $n$-dimensional base space can be parameterized by $n$. The type checker can then prevent operations from accidentally combining objects belonging to incompatible dimensions.

This additional precision comes at a cost. Mathematical expressions that are straightforward on paper may require explicit conversions, equalities between indices, or transport between dependent types in Lean. A significant part of a formalization therefore consists not only of proving mathematical statements, but also of finding representations that expose the relevant invariants without making routine reasoning unnecessarily difficult.

### 2.4.4 Definitions and Theorems

Lean distinguishes between *definitions*, which introduce new objects, and *theorems*, which introduce propositions together with proofs. A simple definition can be written as

```
def square (x : Nat) : Nat :=
x * x
```

while a theorem about this definition can be stated as

```
theorem square_zero : square 0 = 0 := by
rfl
```

The tactic `rfl` proves reflexive equalities. In this example, unfolding the definition of `square` reduces both sides of the equality to the same expression.

More generally, theorem declarations consist of a sequence of parameters and assumptions followed by a proposition to be proved. For instance,

```
theorem add_eq_add_of_eq
(a b c : Nat)
(h : a = b) :
a + c = b + c := by
rw [h]
```

introduces natural numbers $a$, $b$, and $c$, together with an assumption $h : a = b$. The proof then rewrites the occurrence of $a$ using $h$.

Similarly, the assumptions of a CAD rule appear as theorem arguments, while the property established by the rule appears as its result. The formalization chapter will therefore frequently present a mathematical rule together with a corresponding Lean theorem declaration.

### 2.4.5 Proof Terms and Tactics

Although every Lean theorem ultimately requires a *proof term*, users normally construct these terms using *tactics*. Tactics provide a higher-level language for manipulating the current proof state. A *proof state* consists of a collection of assumptions, referred to as the *local context*, and one or more *goals* that remain to be proven.

Consider the theorem

```
theorem transitivity
(P Q R : Prop)
(hpq : P -> Q)
(hqr : Q -> R) :
P -> R := by
intro hp
apply hqr
apply hpq
exact hp
```

Initially, the goal is $P \to R$. The tactic `intro hp` introduces the assumption $hp : P$, leaving $R$ as the goal. The following `apply` commands replace the current goal with the antecedents required by the supplied implications until the assumption $hp$ solves the final goal.

Tactics are therefore a mechanism for *constructing* proofs. They are not, in general, part of the trusted foundation of Lean. Once tactic execution has produced a proof term, the kernel checks that term independently [dMU21]. A bug in a tactic may cause it to fail, produce an inefficient proof, or attempt to construct an invalid term, but an invalid proof term will be rejected by the kernel.

The same applies to the domain-specific automation described in Section 4.4, which performs non-trivial computation when attempting to solve CAD-related proof obligations. Successful tactic execution does not extend the *trusted computing base* merely because the tactic itself performs complex reasoning; the resulting proof term must still be accepted by Lean's kernel.

Conceptually, the process can be viewed as

$$\text{proof obligation} \longrightarrow \text{tactic execution} \longrightarrow \text{proof term} \longrightarrow \text{kernel verification}.$$

### 2.4.6 Simplification and Decision Procedures

Lean provides a variety of proof automation mechanisms. The *simplifier*, invoked using `simp`, repeatedly applies registered rewriting rules to simplify expressions, while `rw` performs explicit rewriting using equalities [AdKU]. Mathlib additionally provides specialized tactics such as `norm_num` for elementary numerical reasoning [Lea20].

For suitable *decidable* propositions, Lean can also evaluate a decision procedure directly. For example,

```
example : (4 : Nat) < 7 := by
decide
```

works because comparison of natural numbers is decidable.

More sophisticated tactics target particular mathematical theories. For example, arithmetic tactics may normalize polynomial expressions or solve systems of linear or nonlinear arithmetic constraints. These tactics can substantially reduce the amount of manual proof code required in a formal development.

Generic automation is nevertheless insufficient for many of the proof obligations generated by the CAD proof system. Statements involving indexed roots, symbolic intervals, or CAD-specific representations have semantic structure that generic Lean tactics do not recognize. Section 4.4 therefore introduces additional domain-specific automation that interprets such objects and reduces them to proof obligations that can be handled by existing Lean mechanisms.

### 2.4.7 Mathlib

Lean's standard library provides the basic language and foundational data structures, while a large part of the mathematical infrastructure used in formal developments is provided by *Mathlib* [Lea20]. Mathlib is a community-developed library containing definitions, theorems, and proof automation covering a broad range of mathematics.

For the work presented in this thesis, Mathlib provides important foundations for algebra, real numbers, sets, topology, and multivariate polynomials. Rather than reimplementing these structures, the formalization builds on the corresponding established definitions wherever possible. Previously proved mathematical results can therefore be reused directly, and the common Mathlib representations keep the development compatible with existing and future developments.

However, Mathlib is a general-purpose mathematical library and does not provide the complete specialized theory required for cylindrical algebraic decomposition. Concepts such as indexed root expressions, symbolic CAD intervals, or the properties and inference rules of the CAD proof system therefore require dedicated definitions and proofs. Moreover, some mathematical results underlying CAD depend on substantial theories of polynomial root behaviour and real algebraic geometry. Whether the required results are already available in a sufficiently suitable form in Mathlib directly influences the amount of additional mathematics that has to be formalized.

### 2.4.8 The Lean Kernel and the Trusted Computing Base

The final correctness check of a Lean theorem is performed by its *kernel*. The kernel implements the small set of rules required to check that a proof term has the claimed type [dMU21]. The theorem prover can therefore contain considerably more complex components—including elaboration, simplification procedures, tactics, and external

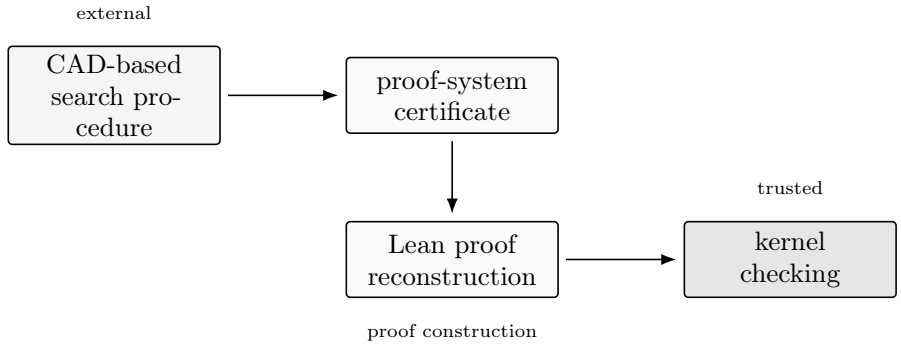



Figure 2.2: Overview of the certificate-checking architecture. An external CAD-based search procedure produces a proof-system certificate. Lean reconstructs a proof from this certificate, while the kernel performs the final trusted correctness check.

proof-search mechanisms—without requiring all of these components to be trusted for logical soundness [dMU21, MMK+25].

This distinction between the trusted kernel and untrusted proof-producing components is what allows the CAD search to remain outside Lean. A CAD-based solver may contain sophisticated heuristics for choosing cells, ordering roots, or selecting projection polynomials, and formal verification of the complete search implementation would require reasoning about all of these algorithmic decisions. In a *proof-producing architecture*, these decisions instead determine which certificate is generated, and the certificate is then checked against formally verified inference rules. Figure 2.2 summarizes the intended architecture.

Only the final proof accepted by the kernel establishes the theorem within Lean. The certificate merely has to contain sufficient information to reconstruct such a proof.

## 2.5 Alethe

SMT solvers combine propositional reasoning with specialized decision procedures for different background theories. While this architecture enables efficient solving, it makes the generation of independently checkable proofs challenging: proof steps may originate from the SAT solver, theory solvers, quantifier instantiation, or preprocessing procedures, and the internal organization of these components differs substantially between solvers. *Alethe* is a proof format designed to provide a common representation for such SMT proofs [SFBF21]. It was defined with the goal of providing a solver-independent interface between proof-producing SMT solvers and proof consumers such as proof checkers and proof assistants.

The syntax of Alethe is deliberately close to SMT-LIB. Terms and formulas largely use the SMT-LIB representation, and existing SMT-LIB constructs are reused where possible [SFBF21, SMT]. An Alethe proof is represented as a sequence of *commands*. The most important commands are `assume`, which introduces an assumption, `step`, which records the application of an inference rule, and `anchor`, which introduces a nested *subproof*. SMT-LIB commands such as `define-fun` may additionally be used, for example to introduce terms occurring during Skolemization [SFBF21].

An ordinary proof step has, schematically, the following form:

```
(step <id> (cl <literal_1> ... <literal_n>)
    :rule <rule>
    :premises (<premise_1> ... <premise_k>)
    :args (...))
```

Here, `<id>` uniquely identifies the proof step and can subsequently be referenced as a *premise*. The expression `cl` represents the *clause* derived by the step. The `:rule` attribute identifies the applied inference rule, while `:premises` refers to previously established proof steps. Some rules additionally carry information through `:args`. Consequently, although an Alethe proof is serialized as a sequence of commands, references between steps describe the dependency structure of the proof.

At its core, Alethe employs a natural-deduction-style calculus in which resolution plays a central role. Additional rules capture reasoning that is specific to SMT solving. Subproofs introduced by `anchor` provide a mechanism for representing reasoning under a local context. This is required, for example, when transformations of bound variables have to be justified locally.

A central design principle of Alethe is that proof rules may be provided at different levels of granularity. A *fine-grained* proof explicitly contains most intermediate reasoning steps and can therefore be checked with comparatively little additional reasoning. A *coarse-grained* proof may leave some of these steps implicit, making the proof easier for a solver to produce at the cost of additional work for the proof checker. This trade-off is intentional: requiring every solver to expose the same amount of internal information would tightly couple the proof format to particular solver implementations. Alethe instead aims to specify the semantics of proof rules while allowing different degrees of detail in their application [SFBF21].

The format is also extensible. To contribute a proof step, a procedure only needs to emit an application of an appropriate rule together with the premises and arguments required by that rule, and the step can be validated independently of the algorithm that produced it. Together with its solver independence and its flexible rule granularity, this makes Alethe a suitable format for integrating additional theory-specific reasoning

into an SMT proof. In this thesis, Alethe is used as the concrete proof representation in which the proof steps introduced later are expressed.

# Chapter 3

# Formalization

The previous chapter introduced the mathematical objects and inference rules of the CAD proof system. This chapter describes their formalization in Lean. The objective is to obtain a representation in which a derivation generated by a CAD-based procedure can be expressed as a sequence of theorem applications and checked by Lean.

A direct transcription of the mathematical presentation is not sufficient. On paper, dimensions, variable sets, projections, and well-formedness conditions are often determined from context. In Lean, we have to represent these dependencies explicitly. We chose not to encode every condition directly in dependent types, since this would make concrete certificates difficult to construct. Instead, dimensions are part of the types of the central objects, while most semantic well-formedness conditions are represented as propositions that occur as premises of the proof rules.

The rule declarations also differ in their current verification status. The development contains interfaces for a large part of the proof system, but not every mathematical result required for their correctness has been proved in Lean. Rules for which the required mathematics is still missing are represented by *axioms*, and several auxiliary results are still *admitted*. The current development is therefore only partially verified and should not be interpreted as an end-to-end verification of the complete proof system.

Section 3.4 records these assumptions explicitly, so that it is clear which parts Lean already checks and which CAD results are still trusted.

## 3.1 Core Representations

### 3.1.1 Levels, Polynomials, Cells, and Sample Points

The fundamental objects are indexed by their dimension. A level is represented by a natural number, while a polynomial of level at most $n$ is represented using Mathlib's multivariate polynomial type [Lea20] with variables indexed by $\mathrm{Fin}(n)$. The basic types are

```
abbrev Level := Nat
abbrev Poly (n : Nat) := MvPolynomial (Fin n) Rat
abbrev Cell (i : Level) := Set (Fin i -> Real)
abbrev SamplePoint (i : Level) := Fin i -> Real
```

Thus, a term of type `Poly i` represents an element of

$$\mathbb{Q}[x_1, \ldots, x_i],$$

and a sample point of level $i$ is a function assigning a real value to every coordinate in $\mathrm{Fin}(i)$. Likewise, a cell is represented extensionally as a set of such points.

This representation makes the dimension part of the type. In particular, a polynomial in $i+1$ variables cannot accidentally be used where a polynomial in $i$ variables is expected. Projection between dimensions therefore has to be explicit as well. For cells, the operation `Cell.proj` existentially eliminates the last coordinate, while `SamplePoint.dropLast` restricts a sample point to its first $i$ coordinates. The final coordinate is accessed separately by `SamplePoint.last`.

The formalization relies on both Mathlib and additional CAD-specific infrastructure. Mathlib provides general algebraic and analytic foundations, including multivariate polynomials, resultants, discriminants, and analytic functions. The module `Cad.Multivariate.ProjectionTheorem` provides notions required for McCallum's projection theory, such as order invariance, non-vanishing, analytic submanifolds, and analytic delineability [McC98].

While the levelwise proof system represents polynomials with rational coefficients, the projection infrastructure operates over the reals. In particular, `PolyR i` represents polynomials as univariate polynomials in the highest variable with coefficients in $\mathbb{R}[x_1, \ldots, x_i]$. The conversion `toMvPolyR` maps rational coefficients to real coefficients, while `toPolyR` additionally converts a polynomial of level $i+1$ into this univariate representation. The latter accounts for the variable ordering introduced in Section 2.1.1, and its compatibility with polynomial evaluation is proved in Lean.

The definitions and inference rules of the levelwise proof system are implemented in `Cad.LevelwiseSCC`, using these supporting definitions where necessary. The projection operations needed by the proof rules are implemented directly on these representations. In particular, a polynomial of level $i+1$ can be regarded as a univariate polynomial in its highest variable with coefficients in $\mathbb{Q}[x_1, \ldots, x_i]$.

From this representation the implementation defines its discriminant, leading coefficient, resultant with another polynomial, coefficients, and factors. Specialization at a sample point produces a univariate real polynomial, whose real roots are represented by

$$\{x \in \mathbb{R} \mid p(x) = 0\}.$$

Order invariance requires slightly more structure. The order of a multivariate polynomial at a point is represented through its partial derivatives. The predicate `firstNonzeroOrder` states that some partial derivative of total order $k$ is non-zero at a point $s$, while every derivative of smaller total order vanishes. The separate predicate `infiniteOrderAt` covers the case in which all partial derivatives vanish. These predicates are subsequently used to express order invariance on a cell.

### 3.1.2 Indexed Root Expressions

A central object in levelwise CAD is an indexed root expression. Mathematically, an indexed root expression refers to the $k$-th real root of a polynomial after specializing its lower variables [NÁS+24]. The Lean representation stores exactly the two pieces of syntactic information required for this reference:

```
structure IndexedRootExpr (i : Level) where
p : Poly (i + 1)
k : Nat
```

The semantic predicate `IsIndexedRealRoot` expresses that $x$ is in fact the $k$-th real root of $p$ above the sample $s$. Besides requiring $k > 0$ and $p(s,x) = 0$, it states that exactly $k - 1$ distinct real roots occur below $x$.

An indexed root expression must also be evaluable when it occurs as a symbolic cell boundary. The function `IndexedRootExpr.eval` therefore returns a real number. If a root satisfying `IsIndexedRealRoot` exists, classical choice selects such a root. Otherwise, the function returns 0.

This is convenient in Lean because it avoids propagating an optional value through every definition, but it has an important consequence: evaluating an indexed root expression only has its intended mathematical meaning once existence of the referenced root has been established.

The formalization therefore separates the syntactic root expression from its well-formedness. In particular, root-ordering predicates such as `matches_sample` require every root expression occurring in an ordering to denote an actual indexed real root at the given sample. An `IndexedRootExpr` thus carries no proof that the referenced root exists; existence is stated as a proposition in the predicates that use the expression rather than being embedded in a dependent type.

### 3.1.3 Symbolic Intervals

Symbolic intervals encode the description of a cell in one coordinate [NÁS+24]. Their finite endpoints are indexed root expressions, while an absent endpoint represents an unbounded direction. The representation additionally records whether either endpoint is open or closed:

```
structure SymbolicInterval (i : Level) where
lower : Sum (Option (IndexedRootExpr i)) Unit
upper : Sum (Option (IndexedRootExpr i)) Unit
lower_closed : Bool
upper_closed : Bool
```

Helper functions expose the finite lower and upper bounds as optional indexed root expressions. Based on these functions, the implementation defines predicates such as `is_bounded`, `is_unbounded`, `is_section`, and `is_sector`. The interpretation of an interval at a point is given by `point_in_interval`.

A section is represented by equal finite lower and upper indexed root expressions with both endpoints closed. All other symbolic intervals are classified as sectors. This provides a uniform interval representation which can also express the closed and half-closed intervals required by the proof system.

### 3.1.4 Indexed Root Orderings

Root orderings record relationships that must remain valid while the underlying cell is generalized [NÁS+24]. Three corresponding structures occur in the formalization.

An ordinary indexed root ordering consists of two relations:

```
structure IndexedRootOrdering (i : Level) where
approx :
Set (Prod (IndexedRootExpr (i + 1))
(IndexedRootExpr (i + 1)))
strict :
Set (Prod (IndexedRootExpr (i + 1))
(IndexedRootExpr (i + 1)))
```

The relation `approx` represents equality of root values, whereas `strict` represents a strict ordering. The predicate `matches_sample` gives these relations their intended semantics at a sample $s$. It first requires every referenced root to exist and subsequently checks

$$\xi(s) = \xi'(s)$$

for pairs in `approx` and

$$\xi(s) < \xi'(s)$$

for pairs in `strict`.

A *weak indexed root ordering* replaces equality by a non-strict ordering relation. Its semantic property therefore requires

$$\xi(t) \leq \xi'(t)$$

for weakly ordered pairs and retains a separate relation for strict inequalities.

*Projective indexed root orderings* are represented separately. In the current development, this representation is intentionally lightweight and stores only the relation required by the corresponding rule interfaces. The complete projective root semantics of the mathematical proof system are not yet encoded; this is part of the current *formalization boundary* discussed in Section 3.4.

## 3.2 Formalization of Cell Properties

The inference rules of the CAD proof system do not operate directly on projection sets. Instead, they derive properties of cells and polynomials. The basic property vocabulary introduced by the proof system contains, among others, sample membership, sign and order invariance, non-nullification, connectedness, the analytic-submanifold property, and analytic delineability. Further properties describe symbolic representations of cells, orderings of indexed roots, non-emptiness, coverings, and eventually truth invariance.

The Lean formalization follows this property-oriented structure. Each property is represented as a proposition whose arguments make the involved dimension, polynomial, sample, or cell explicit. This section discusses the properties that are central either to the mathematical proof system or to the formalization itself. More specialized predicates are introduced where they become relevant, rather than enumerating every auxiliary definition independently.

### 3.2.1 Structural Properties of Cells

The simplest cell property is the *sample property*. For a sample point $s$ and a cell $R$, the proposition

$$\text{sample}(s,R)$$

is defined by

$$s \in R.$$

Although mathematically trivial, this property occurs throughout the rule system. Keeping it as an explicit proposition is useful because membership in a higher-dimensional cell must often be derived from membership in its projection and from satisfaction of the corresponding symbolic interval.

The proof system also uses connectedness and the property of being an analytic submanifold. These assumptions are essential for the mathematical results underlying delineability: roughly speaking, they provide a sufficiently regular domain on which polynomial root functions can be followed consistently.

The current development contains two representations of these properties. Most rule interfaces, including the geometric variants of Rules 4.1, 4.6, 4.8, 4.9, and 4.11, use Mathlib's `IsConnected` and the predicate `IsAnalyticSubmanifold` from `Cad.Multivariate.ProjectionTheorem`, which follows McCallum's local characterization of analytic submanifolds as regular zero sets of analytic maps. The *simplified predicates* of `Cad.LevelwiseSCC` occur far less often. The predicate `connected` expresses path-connectedness through explicitly parametrized paths; together with the derived `simply_connected`, it occurs in Rules 4.30, 4.38, 4.45, and 4.46. The predicate `an_cyl_cell`, used in Rules 4.29–4.30, is defined as `connected` and is thus much weaker than an analytically bounded cylindrical cell. The predicate `an_sub` only adds the trivial condition that every point lies in an open set and thus encodes no charts or analyticity; it occurs only in auxiliary lemmas.

No theorems relate the two representations, so they are not interchangeable: proving a simplified predicate does not establish the stronger property, and rules stated with the stronger predicates cannot be applied to it. A faithful predicate does not make a rule verified either; the rules listed above are axioms, and the proof of Rule 4.30 is immediate only because `an_cyl_cell` coincides with `connected`. Rules using the simplified predicates should therefore be read relative to these Lean definitions.

### 3.2.2 Sign and Order Invariance

Two of the most frequently occurring polynomial properties are sign invariance and order invariance.

Sign invariance is represented by the predicate `sgn_inv`. Instead of first defining a numerical sign function, the implementation introduces a relation `Sign.Same` on real values. For $a$,$b \in \mathbb{R}$, the relation states that

$$a < 0 \iff b < 0, \qquad a = 0 \iff b = 0, \qquad a > 0 \iff b > 0.$$

The proposition

$$\mathrm{sgnInv}(p,R)$$

then requires the values of $p$ at every pair of points in $R$ to satisfy this relation. Thus, a polynomial may be everywhere positive, everywhere negative, or everywhere zero on a sign-invariant cell, but it cannot change between these cases.

This relational representation is convenient for subsequent rules because the logical consequences of sign invariance can be proved directly. For example, equality and strict comparisons with zero are immediately preserved throughout the cell.

Order invariance is represented by `ord_inv`. The implementation avoids introducing a separate order-valued function taking values in $\mathbb{N} \cup \{\infty\}$. Instead, it compares

points through the predicates `firstNonzeroOrder` and `infiniteOrderAt`. For every pair of points in a cell, either the same finite order must be present at both points or both must have infinite order.

### 3.2.3 Nullification and Non-Nullification

A difficulty specific to CAD projection occurs when a polynomial becomes identically zero in its highest variable after specialization of the lower variables. The proof system therefore distinguishes nullification from non-nullification.

The predicate `non_null` expresses that, for every point of the underlying cell, the specialized polynomial is not the zero polynomial as a function of the remaining variable. Operationally, this is represented by requiring the existence of some value of the highest variable at which the polynomial evaluates to a non-zero value.

Thus,

$$\text{nonNull}(p,R)$$

rules out the case in which the complete fibre polynomial vanishes above a point of $R$. This is an important premise of several projection and delineability rules because standard delineability arguments may fail at nullified fibres.

Conversely, `is_nullified` represents the nullified case by requiring the relevant polynomial evaluation to vanish throughout the cell. The proof system contains dedicated rules for handling this situation, for example by reducing sign-invariance obligations to invariance properties of the polynomial coefficients.

### 3.2.4 Delineability

Delineability is one of the central properties of the CAD proof system. At the mathematical level, it ensures that the real roots of a polynomial vary regularly over the underlying cell: roots can be represented by root functions, they remain consistently ordered, and their multiplicity structure does not change. Analytic delineability strengthens this by requiring the root functions to be analytic.

The predicate `an_del` requires that the underlying region is a connected analytic submanifold on which the polynomial is analytically delineable. This means that its real roots can be described by finitely many analytic, strictly ordered root functions with constant multiplicities. Additionally, the polynomial must be order-invariant on each of its root sections.

*Local analytic delineability* differs from this ordinary notion in which roots it constrains. The predicate `an_loc_del` considers only the roots inside the selected symbolic band rather than every real root over the base region. For these roots, it requires analytic, strictly ordered root functions with constant multiplicities and order invariance on the corresponding sections. Whereas the mathematical definition permits a larger surrounding region, the Lean predicate fixes this region to the represented band and is therefore more restrictive.

The predicate `an_proj_del` for *analytic projective delineability* is a simplification: it consists of `an_del` together with order invariance of the leading coefficient. It does not provide the full treatment of projective roots, roots at infinity, or cyclic orderings of projective roots, and thus covers only situations in which ordinary analytic delineability already holds. Rules using this predicate therefore operate in a restricted setting (Section 3.4).

### 3.2.5 Root-Ordering Properties

The symbolic ordering structures introduced in Section 3.1 become cell properties once their relations are required to hold throughout a cell.

The predicate `ir_ord` interprets an ordinary indexed root ordering over a cell. It first requires the distinguished sample to lie in the cell. For every equality pair stored in the ordering, the corresponding indexed root expressions must evaluate to equal values at every point of the cell, while every strict pair must satisfy

$$\xi(t) < \xi'(t).$$

Thus, `ir_ord` turns the relation stored in an `IndexedRootOrdering` into a semantic statement about root functions over an entire region.

A certificate records the relations between indexed root expressions as an indexed root ordering, while the checker must establish through `ir_ord` that these relations are preserved over the underlying cell.

The predicate `weak_ir_ord` performs the analogous task for weak indexed root orderings, interpreting weak pairs using $\leq$. Similarly, `ipr_ord` gives semantics to projective indexed root orderings.

## 3.3 From Inference Rules to Lean Theorems

As discussed in the Lean preliminaries, an inference rule

$$P_1, \ldots, P_k \vdash Q$$

corresponds naturally to a Lean theorem whose arguments contain proofs of $P_1, \ldots, P_k$ and whose result has type $Q$. The source files follow the numbering of the mathematical proof system, making the correspondence between a mathematical rule and its formal counterpart explicit.

For example, the rule for maintaining sign invariance by keeping all roots outside a represented interval is expressed by the declaration

```
axiom rule_4_10
(h_irred : is_irreducible p)
(h_roots_outside : roots_outside p I ord)
(h1 : sample s R.proj)
(h2 : repr I s R)
(h3 : ir_ord ord s R.proj)
(h4 : AnalyticDelineable (toPolyR p) R.proj) :
sgn_inv p R
```

The structure is close to the mathematical inference rule. The parameters $p$, $I$, $s$, $R$, and the indexed root ordering occur as typed Lean objects. Conditions that are written as assumptions preceding the inference rule in the mathematical presentation become explicit theorem arguments, followed by the properties in the antecedent. The conclusion is represented by the return type.

This explicit representation is already useful independently of whether the theorem has been proved. Applying `rule_4_10` to a concrete certificate produces separate proof obligations for irreducibility, the root-ordering condition, sample membership,

the cell representation, preservation of the ordering, and delineability. The certificate checker therefore cannot silently omit a side condition that is present in the declaration.

However, the keyword `axiom` in the example is significant. It means that Lean currently assumes this implication rather than deriving it from previously verified mathematics. Thus, the formalization of the rule's *statement* and the formal verification of its *correctness* are two separate milestones.

### 3.3.1 Coverage of the Rule System

The Lean files cover rule interfaces corresponding to Rules 4.1 through 4.48 of the mathematical proof system presented by Nalbach [Nal25, Chapter 4] and follow the same thematic organization.

Rules 4.1–4.16 comprise the basic McCallum-style proof system, including delineability, non-nullification, sign and order invariance, cell representations, indexed root orderings, connectedness, non-emptiness, and coverings. Among these, basic rules such as the constant case, factor-based sign invariance, sample generalization, and parts of the representation machinery already have explicit Lean proofs. The more substantial projection and delineability results are currently introduced axiomatically.

Rules 4.17–4.18 represent the equational-constraint optimization. Their theorem signatures distinguish between a section defined by the same polynomial and one defined by another polynomial, and contain the required resultant and discriminant conditions. Their correctness is currently assumed.

Rules 4.19–4.20 address nullification. Rule 4.19 is stated as a theorem and reduces sign invariance of a nullified polynomial to sign invariance of its coefficients. Its current proof delegates to an auxiliary lemma whose proof is not yet completed. Rule 4.20, which requires the more involved derivative argument for order invariance, is currently axiomatized.

Rules 4.21–4.22 introduce local delineability and relevance checking. These rules contain some of the most involved interactions between root functions, resultants, and lower-level orderings. Their interfaces are present, but several of the corresponding technical conditions are restricted or abstracted.

Rules 4.23–4.34 cover semi-sign-invariance, weak indexed root orderings, analytically bounded cylindrical cells, and weak versions of connectedness, non-emptiness, and coverings. The elementary implication from sign invariance to semi-sign-invariance is already verified directly. More involved geometric rules remain axiomatized or operate on simplified properties.

Rules 4.35–4.37 cover the compound-bound extension, Rules 4.38–4.46 cover projective delineability and projective orderings, and Rules 4.47–4.48 connect polynomial and indexed-root constraints to truth invariance. In particular, the polynomial variants of Rule 4.47 are proved directly, whereas the indexed-root variants of Rule 4.48 are currently axiomatized.

The rules connecting local and projective delineability to root orderings differ notably in their implementation and verification status. Rule 4.21 is an axiom for restricted analytic local delineability; its premises comprise the selection of relevant polynomial factors, indexed-root-ordering conditions, non-nullification, sample membership and representation, and order invariance of the factors outside the selected set, but unlike the source it requires open interval bounds and a fixed surrounding region. Rule 4.31 is stated as a theorem but derived from Rule 4.21 and thus depends on its axiom; it converts a weak indexed root ordering into an ordinary one using the strict pairs

and equalities obtained from mutually weak pairs, and does not cover one-directional weak relations for roots outside the interval or closed interval bounds. Rules 4.39 and 4.43 are axioms stated with the restricted predicate `an_proj_del`; Rule 4.43 relates the polynomial roots explicitly to the interval bounds, but is restricted to open bounds and ordinary delineability and does not support cyclic projective orderings. Both variants of Rule 4.40 are proved without CAD-specific axioms, but only because `an_proj_del` already includes `an_del`; they do not establish the general passage from projective to ordinary delineability.

Because the rule interfaces are already available, certificates can refer to them before every rule has been proved. An axiom can later be replaced by a theorem of the same type without changing the certificate format or the high-level derivation.

## 3.4 Formalization Boundary

Several predicates in the current Lean development differ from the corresponding mathematical notions introduced in Chapter 2: some are weaker, whereas others, such as `an_loc_del` and `an_proj_del`, are more restrictive. In addition, some premises and supporting results are still missing.

*Restricted predicates* such as `an_loc_del` and `an_proj_del` impose substantive conditions that deviate from the mathematical notions. *Placeholders*, in contrast, impose no condition at all: they are defined as the proposition `True` and can therefore always be discharged using `trivial`. For example, in `rule_4_9`, the premises concerning delineability of the defining polynomials and order-invariance of the relevant resultants are represented by such placeholders. Consequently, the rule can be applied without establishing these mathematical conditions.

The projective extension is only partially represented. A dedicated projective ordering type and corresponding rule declarations exist, but the present data structure does not yet encode the complete cyclic ordering of projective roots or the point at infinity. Together with the restricted predicate `an_proj_del`, this confines the projective rules to a special case, and the strongest of them are axiomatized.

Several auxiliary lemmas also have incomplete proofs. They concern, among other things, the connection between delineability and sign invariance, the use of representations to establish connectedness, simple connectedness, and analytic-submanifold properties, and the treatment of nullification.

The verification status of a rule depends both on its proof and on whether its Lean formulation faithfully represents the mathematical statement. Rules declared as `axiom` are assumed without proof, and even a declaration marked as `theorem` may depend on such axioms or on admitted proofs represented by `sorryAx`. Moreover, a theorem only establishes the statement expressed by its Lean definitions. Simplified predicates and placeholder premises may weaken this statement, while more restrictive predicates may cover only special cases of the mathematical rule. The command `#print axioms` can reveal dependencies on axioms and admitted proofs, but cannot identify discrepancies between the Lean definitions and their intended mathematical meaning.

Consequently, successful replay of a certificate shows that its derivation has been reconstructed correctly from the available declarations, but the resulting theorem may still depend on unverified CAD-specific assumptions. Checking arbitrary CAD certificates soundly against the full mathematical proof system additionally requires proving

the axiomatized rules and admitted lemmas, replacing simplified and placeholder predicates by their intended definitions, and completing the automatic verification of certificate leaves (Section 4.4).

# Chapter 4

# Proof Automation

The previous chapter introduced the Lean formalization of the objects, properties, and inference rules of the levelwise CAD proof system. These definitions make it possible to express CAD derivations in Lean, but applying the rules manually would defeat much of the purpose of a proof-producing decision procedure. A practical integration therefore additionally requires a mechanism that reconstructs a formal proof from a certificate produced by an external CAD implementation.

This chapter develops such a mechanism. Section 4.1 introduces the certificate representation, and Section 4.2 describes how a parsed certificate is replayed by reconstructing each inference-rule application as an application of the corresponding Lean declaration. After an example in Section 4.3, Section 4.4 presents the verification of certificate leaves through our own tactic. The certificate acts as a *proof skeleton*: the Lean term reconstructed from it is checked against the formal rule interfaces introduced in the previous chapter.

## 4.1 Proof Certificates

A run of the levelwise construction naturally produces a directed proof structure. A desired property is reduced by an inference rule to several antecedents, which are recursively reduced until concrete computational obligations remain [NÁS+24].

The implementation uses an Alethe-inspired certificate syntax as its concrete certificate language [SFBF21]. A certificate is parsed as an S-expression and translated into one of four constructors in Lean:

```
inductive AletheStep where
  | assume (id : String) (formula : Sexp)
  | define (id : String) (value : Sexp)
  | step   (id : String) (clause : Sexp)
           (rule : String) (premises : Array String)
           (args : Array Sexp)
  | anchor (stepId : String) (args : Array Sexp)
```

The value stored in the `:rule` field identifies a Lean declaration rather than an inference rule of the Alethe calculus. A certificate step is accepted when that declaration can be instantiated with the listed premises. The implementation therefore

borrows the command-oriented proof representation of Alethe without implementing the Alethe calculus itself.

This design keeps the certificate language independent from the internal implementation of the CAD search. The external solver decides which rule instance to use, while Lean reconstructs the corresponding theorem application.

## 4.2 Certificate Replay

Certificate reconstruction is provided by two *replay tactics*. The tactic `smt_replay` performs *strict* certificate checking, while `smt_replay_dev` is intended for development.

A `define` command is elaborated as a Lean term and introduced as a local definition. This is useful for certificate objects such as polynomials, indexed root expressions, symbolic intervals, sample points, and root orderings. Later certificate steps can refer to these objects by name rather than reproducing their complete representation.

An `assume` command denotes a *certificate leaf*. Unlike a `step` command, it does not apply an inference rule to previously established premises. Instead, it specifies a proposition that must be proved independently by Lean using the `cad_decide` tactic. If the proof succeeds, subsequent certificate steps can use the established proposition. Otherwise, strict certificate replay fails.

A `step` command denotes an *inference step*. The rule name is elaborated as a Lean declaration and applied to the proofs associated with its listed premises. Intermediate results are stored under their certificate identifiers and can subsequently be referenced by later steps.

The normal replay command

```
smt_replay "certificate.alethe"
```

operates strictly and fails if a certificate leaf cannot be established. During development, the alternative

```
smt_replay_dev "certificate.alethe"
```

can be used. If a leaf is unsupported, this variant may replace it with `sorry` after issuing a warning. This permits the replay infrastructure to be tested independently from incomplete leaf automation.

The distinction between these modes is useful during development, but only strict replay provides evidence that every certificate leaf encountered by the reconstruction procedure has been checked.

## 4.3 Example: Verifying a Nonlinear Covering

We illustrate the interaction between certificate replay and the CAD rule system using a two-dimensional covering example. The Lean source contains the base cell, three polynomials, and the final theorem:

```
noncomputable def Rbase2DNonlinear : Cell 1 :=
  { x | And (0 < x 0) (x 0 < 1) }
```

```
noncomputable def qParabola2D : Poly 2 :=
  MvPolynomial.X 1 - MvPolynomial.X 0 ^ 2

noncomputable def qLine2D : Poly 2 :=
  MvPolynomial.X 1 - MvPolynomial.X 0

noncomputable def qRes2D : Poly 1 :=
  MvPolynomial.X 0 - MvPolynomial.X 0 ^ 2

lemma exampleCovering :
    Exists fun s : SamplePoint 1 =>
      Exists fun C : List (SymbolicInterval 2) =>
        cov C s Rbase2DNonlinear := by
  smt_replay
    "Cad/LevelwiseSCC/Examples/ExampleCovering.alethe"
```

Here, `Cell 1` abbreviates `Set (Fin 1 -> Real)` (Section 3.1), representing subsets of $\mathbb{R}^1$. A point `x` is represented as a function from `Fin 1` to `Real`, so `x 0` denotes its only coordinate. Polynomial variables use the same zero-based indexing: `MvPolynomial.X 0` and `MvPolynomial.X 1` represent $x_1$ and $x_2$, respectively.

The Lean theorem itself does not explicitly construct the sample point, the symbolic intervals, or the intermediate CAD properties. These objects and their dependencies are supplied by the accompanying certificate.

### 4.3.1 Geometric Setting

The base cell is

$$R = \{x_1 \in \mathbb{R} \mid 0 < x_1 < 1\}.$$

The level-two polynomials are

$$p(x_1,x_2) = x_2 - x_1^2$$

and

$$q(x_1,x_2) = x_2 - x_1.$$

Their roots with respect to $x_2$ are

$$\theta_p(x_1) = x_1^2, \qquad \theta_q(x_1) = x_1.$$

Since

$$x_1^2 < x_1$$

for every $x_1 \in (0,1)$, the two symbolic intervals

$$I_{\text{lower}} = (-\infty,\theta_q)$$

and

$$I_{\text{upper}} = (\theta_p,\infty)$$

cover the complete cylinder over $R$. The resulting geometry is illustrated in Figure 4.1.

The certificate chooses

$$s_1 = \frac{1}{2}$$

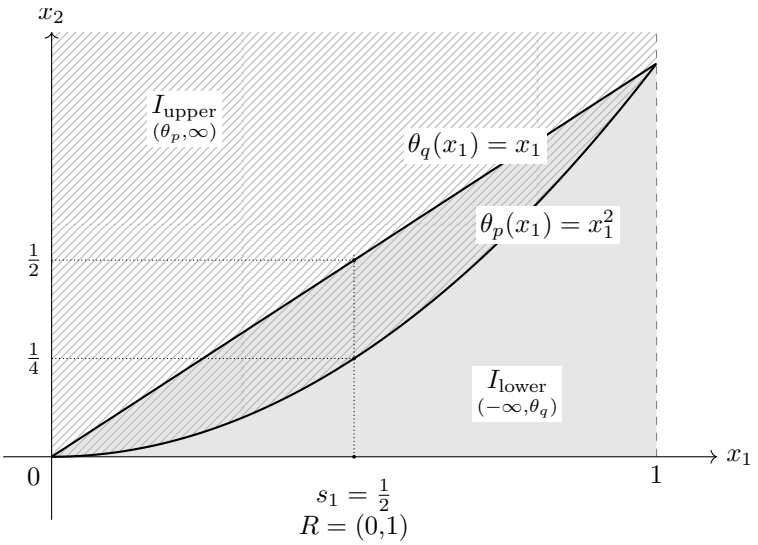



Figure 4.1: Geometry of the nonlinear covering example. Over the base cell $R = (0,1)$, the root functions $\theta_p(x_1) = x_1^2$ and $\theta_q(x_1) = x_1$ satisfy $\theta_p < \theta_q$. Consequently, $I_{\text{lower}} = (-\infty,\theta_q)$ and $I_{\text{upper}} = (\theta_p,\infty)$ overlap and cover the complete cylinder over $R$. At the sample $s_1 = \frac{1}{2}$, the corresponding root values are $\frac{1}{4}$ and $\frac{1}{2}$.

as the sample point. To describe the base cell symbolically, it uses the polynomial

$$r(x_1) = x_1 - x_1^2,$$

whose real roots are 0 and 1. Let

$$\xi_{r,1} \qquad \text{and} \qquad \xi_{r,2}$$

denote the corresponding indexed root expressions. The base interval is then

$$I_{\text{base}} = (\xi_{r,1},\xi_{r,2}),$$

together with the ordering

$$\xi_{r,1} \prec \xi_{r,2}.$$

On the second level, let $\xi_p$ and $\xi_q$ denote the first indexed roots of $p$ and $q$, respectively. At the sample point,

$$\xi_p\left(\frac{1}{2}\right) = \frac{1}{4} < \frac{1}{2} = \xi_q\left(\frac{1}{2}\right).$$

The purpose of the subsequent certificate steps is to extend the corresponding ordering over the complete base cell.

### 4.3.2 Certificate Objects and Leaves

The beginning of the certificate introduces the objects required by the proof:

```
(define s1 ...)
(define s0 ...)

(define xiRes1 ...)
(define xiRes2 ...)
(define Ibase ...)
(define ordBase ...)

(define xiParabola ...)
(define xiLine ...)
(define Ilower ...)
(define Iupper ...)
(define ord2DNonlinear ...)
```

It then provides a number of concrete leaf obligations:

```
(assume j0_nonempty ...)
(assume j1_holds ...)

(assume g0_base_matches ...)
(assume g1_base_ordered ...)
(assume g2_base_sector ...)
(assume g3_base_bounded ...)
(assume g4_point_in_base ...)

(assume g5_cov_matches ...)
(assume g6_lower_ordered ...)
(assume g7_upper_ordered ...)
```

These leaves contain the concrete information required to instantiate the more general CAD rules. Among other things, they establish properties of the symbolic intervals, membership of the sample point, and the indexed-root orderings at the relevant samples.

For example, the ordering of the base roots reduces to

$$\xi_{r,1} = 0, \qquad \xi_{r,2} = 1,$$

and therefore to $0 < 1$. Similarly, the second-level ordering reduces to

$$\xi_p\left(\frac{1}{2}\right) = \frac{1}{4}$$

and

$$\xi_q\left(\frac{1}{2}\right) = \frac{1}{2}.$$

The procedure used to establish such facts is described in Section 4.4.2.

### 4.3.3 Reconstructing the Base Cell

The first certificate steps establish properties of the zero-dimensional projection:

```
(step a0_sample0 (cl)
  :rule rule_4_12_base
  :premises (Rbase2DNonlinear.proj s0 j0_nonempty)
  :args ())

(step a1_ansub0 (cl)
  :rule rule_4_8_base
  :premises (Rbase2DNonlinear.proj j0_nonempty)
  :args ())

(step a2_conn0 (cl)
  :rule rule_4_11_base
  :premises (Rbase2DNonlinear.proj j0_nonempty)
  :args ())
```

These steps establish the sample property, the analytic-submanifold property, and connectedness on the zero-dimensional projection.

The base indexed-root ordering is then obtained by

```
(step b0_irord_base (cl)
  :rule rule_4_9
  :premises
    (g0_base_matches
     a0_sample0
     a1_ansub0
     a2_conn0
     trivial
     trivial)
  :args ())
```

Together with the ordering of the interval bounds, this yields non-emptiness of the base interval:

```
(step b1_nonempty_base (cl)
  :rule rule_4_14
  :premises
    (g0_base_matches
     a0_sample0
     g1_base_ordered
     b0_irord_base)
  :args ())
```

The sector variant of Rule 4.13 [Nal25, Rule 4.13] then turns the concrete interval description into a representation of the base cell:

```
(step b2_repr_base (cl)
  :rule rule_4_13_sector
  :premises
    (g2_base_sector
     a0_sample0
     j1_holds
     b1_nonempty_base)
  :args ())
```

Using this representation, the sample point $s_1 = 1/2$ can be lifted to the one-dimensional cell:

```
(step b3_sample1 (cl)
  :rule rule_4_12_ind
  :premises
    (g4_point_in_base
     a0_sample0
     b2_repr_base)
  :args ())
```

The remaining properties required for the next level are established by

```
(step b4_ansub1 (cl)
  :rule rule_4_8_ind
  :premises
    (b2_repr_base
     a1_ansub0)
  :args ())

(step b5_conn1 (cl)
  :rule rule_4_11_bounded
  :premises
    (g3_base_bounded
     g1_base_ordered
     a2_conn0
     b2_repr_base
     a0_sample0
     b0_irord_base)
  :args ())
```

The proof has thus reconstructed the required properties of

$$R = (0,1)$$

from properties of its zero-dimensional projection and the symbolic interval used to describe it. This illustrates the levelwise structure of the proof system directly.

### 4.3.4 Establishing the Covering

The second part of the certificate concerns the ordering

$$\xi_p \prec \xi_q.$$

Using the previously established properties of the base cell, the certificate obtains

```
(step c0_irord_cov (cl)
  :rule rule_4_9
  :premises
    (g5_cov_matches
     b3_sample1
     b4_ansub1
```

```
     b5_conn1
     trivial
     trivial)
  :args ())
```

This ordering allows the two intervals forming the covering to be shown non-empty:

```
(step c1_nonempty_lower (cl)
  :rule rule_4_14
  :premises
    (g5_cov_matches
     b3_sample1
     g6_lower_ordered
     c0_irord_cov)
  :args ())

(step c2_nonempty_upper (cl)
  :rule rule_4_14
  :premises
    (g5_cov_matches
     b3_sample1
     g7_upper_ordered
     c0_irord_cov)
  :args ())
```

The helper theorem `all_nonempty_two` combines the two results into the form required by the covering rule:

```
(step c3_all_nonempty (cl)
  :rule all_nonempty_two
  :premises
    (c1_nonempty_lower
     c2_nonempty_upper)
  :args ())
```

Likewise, `covering_conditions_two` combines the relevant matching and ordering conditions:

```
(step c4_cov_conditions (cl)
  :rule covering_conditions_two
  :premises
    (g5_cov_matches
     g6_lower_ordered
     g7_upper_ordered)
  :args ())
```

Rule 4.16 [Nal25, Rule 4.16] can then establish the covering:

```
(step c5_cov (cl)
  :rule rule_4_16
  :premises
    (c4_cov_conditions
```

```
    b3_sample1
    c0_irord_cov
    c3_all_nonempty)
  :args ())
```

Its conclusion is

$$\mathrm{cov}\left([I_{\mathrm{lower}},I_{\mathrm{upper}}], s_1, R\right).$$

The original Lean theorem only requires the existence of some sample point and some covering. The final certificate step therefore applies

```
(step final (cl)
  :rule CAD.Decide.exists_covering_of_cov
  :premises (c5_cov)
  :args ())
```

and obtains

$$\exists s\, C.\ \mathrm{cov}(C,s,R).$$

Ignoring the implementation details of the leaves, the complete derivation can be summarized as

$$\begin{array}{c}
\text{base interval facts} \\
\Downarrow \\
\mathrm{ir_ord}_{\mathrm{base}} \Longrightarrow \mathrm{non_empty}(I_{\mathrm{base}}) \Longrightarrow \mathrm{repr}(I_{\mathrm{base}}) \\
\Downarrow \\
\mathrm{sample}(s_1) \wedge \mathrm{connected}(R) \wedge \mathrm{an_sub}(R) \\
\Downarrow \\
\xi_p \prec \xi_q \\
\Downarrow \\
\mathrm{non_empty}(I_{\mathrm{lower}}) \wedge \mathrm{non_empty}(I_{\mathrm{upper}}) \\
\Downarrow \\
\mathrm{cov}\left([I_{\mathrm{lower}},I_{\mathrm{upper}}], s_1, R\right).
\end{array}$$

The example therefore shows how a certificate can encode the high-level choices of a CAD computation while leaving the concrete side conditions to a small verifier.

## 4.4 Verification of Certificate Leaves

The tactic `cad_decide` handles the concrete obligations represented by Alethe `assume` commands. It is not intended to perform high-level CAD proof search; that structure is already contained in the certificate.

Its interface is small:

```
def verifyCertificateLeaf (g : MVarId) : TacticM Unit :=
  dischargeAssume g

syntax (name := cadDecide) "cad_decide" : tactic
```

```
@[tactic cadDecide] def evalCadDecide : Tactic := fun _ => do
  verifyCertificateLeaf (← getMainGoal)
  replaceMainGoal []
```

The function `dischargeAssume` examines the target proposition and dispatches to a specialized verification procedure. This allows support for individual kinds of certificate leaves to be extended independently.

### 4.4.1 Structural and Geometric Conditions

Many certificate leaves are finite structural statements about symbolic intervals or concrete cells. Examples include

$$\text{bounds_ordered}(I,\mathcal{O}), \qquad \text{is_sector}(I), \qquad \text{is_bounded}(I).$$

For concrete certificate data, these properties can usually be established by unfolding the relevant definitions and applying simplification, numerical normalization, or decidability.

The checker uses a similar approach for equalities between concrete certificate objects and projections of cells. Where necessary, extensionality is applied before reducing the remaining arithmetic or set-theoretic statement.

The implementation also contains explicit support for basic geometric properties of the one-dimensional open intervals occurring in the current examples. For these intervals, connectedness is established using convexity. Openness together with a concrete non-emptiness witness is used to establish the current analytic-submanifold predicate.

These procedures do not reproduce the higher-level CAD reasoning. Their purpose is only to establish the concrete premises required by the corresponding inference rules.

### 4.4.2 Indexed Root Expressions and Root-Based Conditions

Indexed root expressions require additional reasoning because they encode both a root value and its position in the ordered set of real roots. To prove

$$\xi(s) = r,$$

it is therefore not sufficient to establish

$$p(s,r) = 0.$$

The checker must additionally show that $r$ has the root index stored in $\xi$.

For a polynomial that reduces to

$$p(y) = y - r,$$

the implementation provides a helper result characterizing its first indexed real root:

$$\text{IsIndexedRealRoot}(p,s,1,x) \quad \Longleftrightarrow \quad x = r.$$

This establishes both the root value and the fact that it is the first real root.

A second family of helper results handles polynomials of the form

$$p(y) = (y - r_1)(r_2 - y), \qquad r_1 < r_2.$$

These establish that $r_1$ and $r_2$ are the first and second indexed roots, respectively. In particular, proving that $r_2$ is the second root requires showing that exactly one real root lies strictly below it.

The polynomial

$$r(x_1) = x_1 - x_1^2$$

from Section 4.3 is handled in this way, yielding

$$\xi_{r,1} = 0, \qquad \xi_{r,2} = 1.$$

The procedure `proveConcreteIndexedRootFacts` applies these results to indexed root expressions occurring in certificate leaves. It first extracts the defining polynomial and root index and then attempts to prove the corresponding concrete root statement. The resulting evaluation and existence facts are inserted into the local context before the surrounding CAD predicate is unfolded.

This ordering is important because objects introduced by certificate `define` commands occur as local Lean definitions. Aggressive simplification at the beginning may unfold polynomial lifting and renaming operations far beyond the structure relevant to the proof. The checker therefore extracts the indexed-root information first and applies ordinary simplification afterwards.

Once the root evaluations are available, several CAD-specific predicates reduce to elementary arithmetic and structural conditions. This applies in particular to

$$\text{matches_sample}, \qquad \text{point_in_interval}, \qquad \text{holds}\,.$$

For example, consider a symbolic interval

$$I = (\xi_l, \xi_u).$$

After establishing

$$\xi_l(s) = l \qquad \text{and} \qquad \xi_u(s) = u,$$

checking whether the relevant sample coordinate lies inside the interval reduces to

$$l < s_i < u.$$

Likewise, checking whether an indexed root ordering matches a sample reduces to evaluating the roots occurring in the ordering and comparing their values.

The checker therefore first discovers the indexed root expressions required by a leaf and proves their evaluations. It then unfolds the enclosing CAD predicate and discharges the remaining inequalities and finite structural conditions using simplification and arithmetic tactics.

The current indexed-root verifier supports the concrete polynomial forms required by the prototype examples. A general implementation must also justify computations involving real algebraic numbers, such as root isolation, polynomial evaluation, and comparisons between roots. The correctness of these computations must be established either through additional certificates containing sufficient information to verify the results or through formally verified procedures in Lean. The subsequent treatment of root-based CAD predicates could remain unchanged.

### 4.4.3 Sign-Invariance

Sign-invariance connects computational polynomial reasoning to the semantic CAD properties. The formalization contains *bridge lemmas* of the form

$$(\forall s \in R.\ 0 < p(s)) \Longrightarrow \text{sgn_inv}(p,R)$$

and

$$(\forall s \in R.\ p(s) < 0) \Longrightarrow \text{sgn_inv}(p,R),$$

as well as direct results for zero and constant polynomials.

The current automatic dispatcher handles these elementary cases. Expressions involving operations such as discriminants, resultants, and leading coefficients are first simplified so that constant cases can also be recognized after normalization.

For genuinely non-constant univariate polynomials, the implementation contains the function `proveSignOnGap`, adapted from the univariate decision procedure in `Cad.Univariate`. Given a non-zero rational polynomial, a sorted list of its real roots with rational isolating intervals, and a proof that this list contains all of its roots, it chooses a rational point $m$ with $l < m < r$ between two consecutive roots $l < r$, evaluates the polynomial at $m$, and constructs a Lean proof that the polynomial has this sign throughout $(l,r)$.

## 4.5 Trust Boundary and Current Limitations

The current implementation separates two different sources of incompleteness.

The first concerns certificate leaves. In strict mode, an `assume` command is accepted only if `cad_decide` constructs a proof. Unsupported leaves therefore cause replay to fail. The development variant may temporarily replace such leaves with `sorry`, but the resulting proof is explicitly outside the strictly checked fragment.

The second concerns the mathematical CAD rules themselves. Several of the more involved rules used by the current certificates are still represented axiomatically. Successful strict replay therefore shows that the certificate has been reconstructed correctly relative to the available Lean declarations, but it does not imply that the resulting theorem is free of axioms.

The axioms used by a reconstructed theorem can be inspected directly using

```
#print axioms exampleCovering
```

which reports the axioms on which the theorem depends.

The remaining gaps in the leaf checker include general support for non-trivial sign-invariance, order-invariance, non-nullification, and factorization claims, as well as arbitrary indexed-root computations. The current covering example additionally relies on axiomatized versions of several CAD rules. In particular, `rule_4_9` represents premises concerning delineability of the defining polynomials and order-invariance of the relevant resultants by trivial propositions.

The two kinds of missing support occur in different parts of the checker: concrete computations at certificate leaves and mathematical assumptions in the rule interfaces. They can therefore be addressed separately, by extending cad_decide or by proving the corresponding CAD rules.

# Chapter 5

# Conclusion

In this thesis, we investigated how the levelwise CAD proof system can be formalized and checked in Lean, without implementing a complete CAD algorithm inside the theorem prover. The implementation has two main parts. First, we represented the mathematical objects and inference rules of the proof system in Lean. Second, we implemented a certificate-replay mechanism that applies these rules and proves concrete side conditions inside Lean. Together, these parts show how a proof produced during a CAD computation can be reconstructed in an interactive theorem prover.

## 5.1 Summary

We represented the central objects of the levelwise CAD proof system in Lean, including polynomials, cells, sample points, indexed root expressions, symbolic intervals, and indexed root orderings. Their dimensions are made explicit in the corresponding types, while most conditions are represented as separate propositions and checked as explicit proof obligations.

The inference rules are represented as Lean declarations whose premises and conclusions correspond to the cell properties of the mathematical proof system. The current development contains interfaces for the different groups of rules, including sign and order invariance, cell representations, root orderings, equational constraints, weak orderings, coverings, and truth invariance. Several simpler rules have been proved directly in Lean. More difficult rules, especially those that depend on delineability, analytic properties of cells, or deeper results about polynomial roots and projection polynomials, still rely on axioms, or simplified semantic predicates. The complete CAD calculus is therefore not yet machine-checked, but the remaining assumptions are made explicit.

Concrete proof obligations arising from rule applications can be automated to a useful extent. The `cad_decide` tactic handles structural properties of symbolic intervals, elementary geometric and arithmetic conditions, and selected computations involving indexed root expressions. In particular, indexed-root verification establishes both the value of a root and its position among the real roots of the defining polynomial, after which properties such as interval membership and matching of root orderings often reduce to elementary inequalities. The development also contains the core argument required for determining the sign of a univariate polynomial on a root-free interval. Arbitrary real-root isolation, and automatic verification of non-trivial sign

invariance, order invariance, non-nullification, irreducibility, and related properties remain incomplete.

The nonlinear covering example exercises these components across several levels of the proof system: symbolic objects are introduced by a certificate in the Alethe-inspired syntax [SFBF21], properties of the base cell are reconstructed, the relevant root ordering is established, and the covering property is finally derived. Strict replay guarantees that every encountered certificate leaf has been proved before it is used.

The current checker is therefore still a prototype and does not yet verify arbitrary CAD certificates. Strict replay shows that a certificate has been reconstructed correctly from the available Lean declarations, but the resulting theorem may still depend on axiomatized CAD rules or admitted lemmas. The example thus establishes that a derivation spanning several levels of the proof system can be replayed with all of its certificate leaves verified automatically, but not yet that its conclusion holds independently of these assumptions.

## 5.2 Future Work

Two different sources of incompleteness remain. The first concerns the mathematical formalization. Several of the most substantial results required by the proof system are not yet proved from Lean's existing foundations and therefore remain part of the trusted basis of the current development. The second concerns proof automation. The present cad_decide implementation recognizes the concrete forms required by the prototype certificates, but it is not yet a general decision procedure for every leaf property that may be generated by the levelwise calculus.

The most immediate continuation of this work is therefore to reduce the trusted mathematical basis of the formalization. The axiomatized rules should progressively be replaced by Lean proofs. This will require formalizing substantial parts of the real-algebraic theory underlying CAD, in particular results concerning delineability, continuity and ordering of polynomial root functions, resultants and discriminants, and the topological properties of the cells on which these results are applied.

A related task is to strengthen the semantic definitions used by the current formalization where necessary. Some definitions were chosen with the scope of the prototype in mind. Before arbitrary certificates produced by a general CAD implementation can be claimed to be fully verified, the formal definitions and the mathematical semantics expected by the original proof system need to coincide closely enough that no additional assumptions are hidden in the translation.

This should be accompanied by a general certified interface to real-root isolation. Instead of recognizing a small number of concrete polynomial shapes, future certificates could provide sufficient information to verify isolating intervals and root indices for arbitrary univariate polynomials. Indexed root expressions could then be evaluated by a uniform procedure. The same applies to evaluations and comparisons involving real algebraic numbers, such as determining the sign of a polynomial at an algebraic root. Such computations must either be performed by verified procedures in Lean or be accompanied by certificates that allow their results to be checked independently. Further extensions to cad_decide should focus on concrete certificate leaves, such as polynomial identities or factorization claims while general cell properties are established through the inference rules.

Another important next step is tighter integration with an actual CAD implemen-

tation. The present project primarily establishes the certificate-consumer side of such an interface. A complete proof-producing workflow additionally requires the solver to emit certificates containing the definitions, rule applications, dependencies, and computational witnesses expected by the Lean checker. Establishing this end-to-end connection would make it possible to verify certificates generated from realistic solver runs rather than manually prepared examples.

Such an integration would also enable a systematic evaluation of the approach. In addition to determining whether certificates can be replayed successfully, future experiments should investigate certificate size, replay time, and the cost of checking different categories of leaves. Of particular interest is the trade-off between information supplied by the external solver and computation repeated by Lean. A small certificate places more reconstruction work on the checker, whereas a more explicit certificate can provide root intervals, sign information, factorization results, or other witnesses directly.

The certificate format itself may also have to be extended: the Alethe-inspired representation employed in this thesis is sufficient to demonstrate the replay architecture, but larger CAD proofs may benefit from more structured representations for algebraic numbers, multivariate polynomials, symbolic intervals, and root orderings.

Beyond completing the current implementation, the formalization can also follow further developments of the levelwise proof system itself. Its rule-based organization makes such extensions straightforward. New CAD optimizations can be integrated by introducing the corresponding properties and proving their inference rules, after which an external solver can reference these rules in exactly the same way as the existing ones.

A fully verified certificate checker still requires Lean proofs of the remaining axiomatized rules, semantic definitions that coincide with those of the original proof system, and general verified procedures in place of the specialized leaf checkers. Checking certificates from realistic solver runs additionally requires a CAD implementation that emits them.

# Acknowledgements

I would like to thank my advisors, Jasper Nalbach and Valentin Promies, for their guidance and feedback throughout this thesis. I am also grateful to Prof. Dr. Erika Ábrahám for giving me the opportunity to carry out my master's thesis at THS, and to Prof. Dr. Jürgen Giesl for serving as the second examiner.

# Bibliography


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

[AdKU] Jeremy Avigad, Leonardo de Moura, Soonho Kong, and Ullrich. Theorem Proving in Lean 4. https://lean-lang.org/theorem_proving_in_lean4/.

[Bro01] Christopher W. Brown. Improved Projection for Cylindrical Algebraic Decomposition. *Journal of Symbolic Computation*, 32(5):447–465, November 2001.

[Col75] George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decompostion. In H. Brakhage, editor, *Autom. Theory Form. Lang.*, pages 134–183, Berlin, Heidelberg, 1975. Springer.

[dMU21] Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In André Platzer and Geoff Sutcliffe, editors, *Autom. Deduc. – CADE 28*, pages 625–635, Cham, 2021. Springer International Publishing.

[JBd13] Dejan Jovanovic, Clark Barrett, and Leonardo de Moura. The design and implementation of the model constructing satisfiability calculus. In *2013 Form. Methods Comput.-Aided Des.*, pages 173–180, October 2013.

[Lea20] The lean mathematical library. In *Proc. 9th ACM SIGPLAN Int. Conf. Certif. Programs Proofs*, CPP 2020, pages 367–381, New York, NY, USA, January 2020. Association for Computing Machinery.

[McC98] Scott McCallum. An Improved Projection Operation for Cylindrical Algebraic Decomposition. In Bob F. Caviness and Jeremy R. Johnson, editors, *Quantifier Elimin. Cylind. Algebr. Decompos.*, pages 242–268, Vienna, 1998. Springer.

[MMK+25] Abdalrhman Mohamed, Tomaz Mascarenhas, Harun Khan, Haniel Barbosa, Andrew Reynolds, Yicheng Qian, Cesare Tinelli, and Clark Barrett. Lean-smt: An SMT Tactic for Discharging Proof Goals in Lean. In Ruzica Piskac and Zvonimir Rakamarić, editors, *Comput. Aided Verification*, pages 197–212, Cham, 2025. Springer Nature Switzerland.

[Nal25] Jasper Kurt Ferdinand Nalbach. *Cylindrical Algebraic Decomposition Based Methods in Satisfiability modulo Non-Linear Real Arithmetic*. PhD thesis, RWTH Aachen University, Aachen, 2025.

[NÁS+24] Jasper Nalbach, Erika Ábrahám, Philippe Specht, Christopher W. Brown, James H. Davenport, and Matthew England. Levelwise construction of a single cylindrical algebraic cell. *J. Symb. Comput.*, 123(C), July 2024.

[NK26] Jasper Nalbach and Gereon Kremer. Extensions of the Cylindrical Algebraic Covering Method for Quantifiers. *Math.Comput.Sci.*, 20(1):7, April 2026.

[SFBF21] Hans-Jörg Schurr, Mathias Fleury, Haniel Barbosa, and Pascal Fontaine. Alethe: Towards a Generic SMT Proof Format (extended abstract). *Electron. Proc. Theor. Comput. Sci.*, 336:49–54, July 2021.

[SMT] SMT-LIB The Satisfiability Modulo Theories Library. https://smt-lib.org/language.shtml.