A Variant of Non-uniform Cylindrical Algebraic Decomposition for Real Quantifier Elimination[⋆]

Jasper Nalbach[1,*], Erika Ábrahám[1]

[1]*RWTH Aachen University, Aachen, Germany*

Abstract

The *Cylindrical Algebraic Decomposition (CAD) method* is currently the only complete algorithm used in practice for solving real-algebraic problems. To ameliorate its doubly-exponential complexity, different *exploration-guided* adaptations try to avoid some of the computations. The first such adaptation named *NLSAT* was followed by *Non-uniform CAD (NuCAD)* and the *Cylindrical Algebraic Covering (CAlC)*. Both NLSAT and CAlC have been developed and implemented in SMT solvers for satisfiability checking, and CAlC was recently also adapted for quantifier elimination. However, NuCAD was designed for quantifier elimination only, and no complete implementation existed before this work. In this paper, we present a novel variant of NuCAD for both real quantifier elimination and SMT solving, provide an implementation, and evaluate the method by experimentally comparing it to CAlC.

Keywords

Real Quantifier Elimination, Cylindrical Algebraic Decomposition, SMT Solving.

1. Introduction

Real algebra is a highly expressive logic, whose formulas are Boolean combinations of polynomial constraints over potentially quantified real-valued variables. In this paper, we consider three related problems for real algebra: (1) deciding the satisfiability of quantifier-free formulas (i.e., solving the existential fragment), (2) deciding the truth of sentences, and (3) quantifier elimination.

The first practically feasible complete algorithm for solving all these problems was the *cylindrical algebraic decomposition (CAD) method* [11]. Though doubly-exponential in complexity, CAD and some of its variants are till today the only complete solutions offered in computer algebra tools like `QEPCAD B` [4, 5], `Redlog` [26], and commercial systems like `Mathematica` [27] or `Maple` [10, 16].

On another research line, *exploration-guided proof generation* turned out to be extremely effective for checking the satisfiability of propositional logic formulas via SAT solvers. Extending this idea to *Satisfiability Modulo Theories (SMT) solving*, the *NLSAT* [17] algorithm was the first exploration-guided CAD adaptation for satisfiability checking. Whereas CAD analyses the whole state space regarding the whole input formula, NLSAT first explores the state space through smart guesses of sample values for the (real and Boolean) variables. If a partial sample turns out not to be further extensible to a satisfying assignment, then some partial CAD computations are applied to generalize this sample to a set of samples, which are not extensible to satisfying solutions for the same reason (i.e., the same constraints in the input formula). These generalizations rely on the *single cell* paradigm introduced in [6, 9] and refined in [22]. NLSAT implementations exist in the SMT solvers `z3` [14], `yices2` [15], and `SMT-RAT` [12].

NLSAT inspired the development of another CAD adaptation named the *Cylindrical Algebraic Covering (CAlC)* [1] algorithm. It also uses exploration and single-cell generalization, but it reduces the bookkeeping effort of NLSAT (to remember all the generalizations). Though the original CAlC algorithm

10th International Workshop on Satisfiability Checking and Symbolic Computation, August 2, 2025, Stuttgart, Germany

⋆Jasper Nalbach was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as part of RTG 2236 *UnRAVeL* and AB 461/9-1 *SMT-ART*. Computations were performed with computing resources granted by RWTH Aachen University under project rwth1560.

*Corresponding author.

nalbach@cs.rwth-aachen.de (J. Nalbach); abraham@cs.rwth-aachen.de (E. Ábrahám)
0000-0002-2641-1380 (J. Nalbach); 0000-0002-5647-6134 (E. Ábrahám)

cannot handle Boolean structures other than conjunctions and thus this task needs to be outsourced to a SAT solver, such a (lazy SMT) CAlC implementation in `cvc5` [19] outperforms NLSAT in `z3` and `yices2`.

While having a focus on satisfiability checking, some SMT solvers can also solve quantified formulas [2, 13, 23]. While most such support is incomplete for the theory of the reals, the *QSMA* [3] algorithm implemented in `yicesQS`, which relies on NLSAT, is complete. Recently, also the CAlC algorithm was extended to transfer the performance gains from the SMT world to quantifier elimination [18, 21], successfully outperforming (in terms of running time) existing SMT solvers and open source implementations supporting quantifier elimination.

Also the *non-uniform cylindrical algebraic decomposition (NuCAD)* [7] builds on the NLSAT idea to decompose $\mathbb{R}^n$ similar to the CAD, but with fewer cells. Its first version was not able to reason about quantifier alternations. The work in [8] suggests to first compute a NuCAD of $\mathbb{R}^n$, and then to refine this decomposition to respect the quantifier structure of the input formula. NuCAD as introduced in these papers is incomplete in the sense that it considers only *open cells*, i.e., it might not be able to solve formulas with non-strict inequalities.

Our paper continues the above work on the NuCAD, contributing:

- a *complete* variant of the NuCAD algorithm,
- a modification which considers the input's quantifier structure *directly* during the computation of the decomposition,
- its *implementation* along with a *post-processing* of the NuCAD, and
- *experimental evaluation* in particular against the CAlC algorithm, on deciding the existential fragment, deciding sentences, and quantifier elimination.

Though our experiments indicate that CAlC solves satisfiability checking problems faster than NuCAD, NuCAD seems to be competitive on quantifier elimination. However, the available quantified benchmark sets lack sufficient diversity to make definitive conclusions on algorithm superiority.

Outline: We introduce the fundamentals of CAD and CAlC in Section 2, and introduce a complete version of the NuCAD algorithm in Section 3. In Section 4, we extend the algorithm for quantifier alternations. Finally, we present the experimental results in Section 5 and conclude the paper in Section 6.

2. Preliminaries

Let $\mathbb{N}$, $\mathbb{N}_{>0}$, $\mathbb{Q}$, and $\mathbb{R}$ denote the natural (incl. 0), positive integer, rational, and real numbers respectively. For $i, j \in \mathbb{N}$, we set $[i..j] = \{i, \dots, j\}$ and $[i] = [1..i]$. For $j \in \mathbb{N}$, $s \in \mathbb{R}^j$ and $k \in [j]$ we write s_k for the kth component of s. For tuples introduced as $t = (a, b, c)$, we use $t.a$, $t.b$ and $t.c$ to access their entries.
From now $n \in \mathbb{N}_{>0}$, $i \in [0..n]$,$j \in [n]$, $s \in \mathbb{R}^j$, $s^* \in \mathbb{R}$, $R \subseteq \mathbb{R}^j$, and $I \subseteq \mathbb{R}$.

We define $(s, s^*) = (s_1, \dots, s_j, s^*)$, $s \times I = \{s\} \times I$, $s_{[i]} = (s_1, \dots, s_{\min\{i,j\}})$, and $R_{[i]} = \{s_{[i]} | s \in R\}$. Let $f, g : R \to \mathbb{R}$ be continuous functions, then $R \times (f, g) = \{(r, r') \mid r \in R \wedge r' \in \mathbb{R} \wedge f(r) < r' < g(r)\}$, $R \times [f, g] = \{(r, r') \mid r \in R \wedge r' \in \mathbb{R} \wedge f(r) \leq r' \leq g(r)\}$, analogously $R \times (f, g]$ and $R \times [f, g)$.

Throughout this paper, we assume ordered *variables* $x_1 \prec \dots \prec x_n$. Let $\mathbb{Q}[x_1, \dots, x_i]$ be the set of all *polynomials* in $x_1, \dots, x_i$ with rational coefficients. Let $p \in \mathbb{Q}[x_1, \dots, x_i]$. The *main variable* of p is the highest ordered variable occurring in it, and its *level* $\text{level}(p)$ is the index of the main variable. The *degree* of p in x_j is denoted by $\deg_{x_j}(p)$. For a univariate $p \in \mathbb{Q}[x_j]$, its *real roots* (in x_j) build the set $\text{realRoots}(p)$.

Let $p(s)$ result from p by substituting the values s_k for x_k, for $k \in [i]$. We call p *sign-invariant* on $R' \subseteq \mathbb{R}^i$ if $p(s')$ has the same sign for all $s' \in R'$. The polynomial p is *nullified over* s if $p(s) \equiv 0$, where $\equiv$ denotes semantic equivalence. We further define $p[s]$ to be $p(s)$ for $\text{level}(p) \leq j$, and p otherwise.

A (polynomial) *constraint* has the form $p \sim 0$ with $\sim \in \{=, \leq, \geq, \neq, <, >\}$. Notations for polynomials are transferred to constraints where meaningful, for example $(p \sim 0)(s)$ is $p(s) \sim 0$, and $(p \sim 0)[s]$ is $p[s] \sim 0$.

Real-algebraic *formulae* φ are potentially quantified Boolean combinations of polynomial constraints. Again, we inherit notations from constraints, e.g. we get $\varphi(s)$ ($\varphi[s]$) from φ through replacing each constraint c by $c(s)$ ($c[s]$). A formula φ is *truth-invariant* on $R' \subseteq \mathbb{R}^i$ if $\varphi(s') = \varphi(s'')$ for all s', $s'' \in R'$. A formula φ is in *prenex normal form* if it is of the form $Q_{k+1}x_{k+1} \ldots Q_n x_n.\ \bar{\varphi}$ for some $k \in \mathbb{N}$, consisting of a *prefix* of quantifiers and a quantifier-free formula $\bar{\varphi}$ called the *matrix*; the variables $x_1, \ldots, x_k$ are called *free* in φ (also called *parameters*). The task of *quantifier elimination* is, given a formula φ that may contain quantifiers, to compute a quantifier-free formula ψ such that $\psi \equiv \varphi$.

We will further make use of the following concept:

Definition 2.1 (Implicant [18, 21]). *Let $j \in \mathbb{N}$, $s \in \mathbb{R}^j$, and φ be a formula where (at least) the variables $x_1, \ldots, x_j$ are free.*

The quantifier-free formula ψ is an implicant of φ with respect to s *if all constraints $p \sim 0$ in ψ have* level$(p) \leq j$ *and are contained in φ, $\psi[s] \equiv$ true, and either $\psi \Rightarrow \varphi$ or $\psi \Rightarrow \neg\varphi$.* □

Example 2.1. *As a univariate example with $j = 1$, for $\varphi = x_1^2 > 0 \wedge (x_1 < 2 \vee x_1 > 4)$ we have $\varphi[1] \equiv$ true, $\varphi[3] \equiv$ false, and $\varphi[0] \equiv$ false. Examples for implicants of φ are $x_1^2 > 0 \wedge x_1 < 2$* w.r.t. *the value $s = 1 (\in \mathbb{R}^1)$, $\neg(x_1 < 2 \vee x_1 > 4)$* w.r.t. *3, and both $\neg(x_1^2 > 0)$ and $\neg(x_1^2 > 0 \wedge x_1 > 4)$ are implicants of φ* w.r.t. *0.*

For $\varphi' = (x_1 < 0 \vee x_2 \leq 4) \wedge (x_1 > 2 \vee x_2 > 4)$ we observe $\varphi'[1] \equiv$ false, and identify $\neg(x_1 < 0) \wedge \neg(x_1 > 2)$ to be an implicant of φ' w.r.t. *1.* ⌟

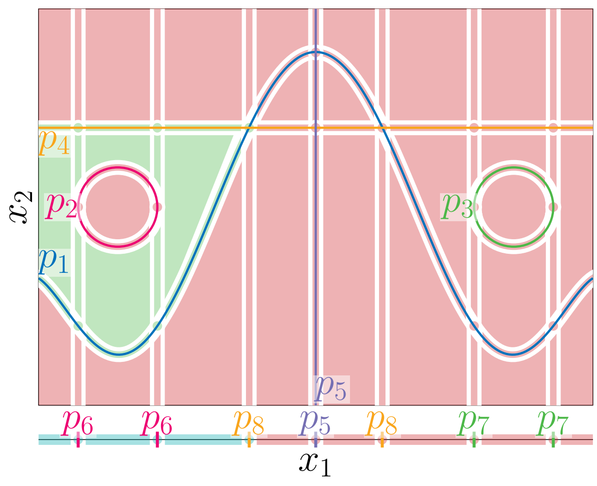
2.1. Cylindrical Algebraic Decomposition and Covering

For a given formula, the *cylindrical algebraic decomposition (CAD)* [11] method decomposes $\mathbb{R}^n$ into a finite number of subsets such that each polynomial in the formula is sign-invariant on each subset. The finiteness of the decomposition allows to reason about the solution space of the formula. Furthermore, to admit quantifier elimination, these sets are arranged in a cylindrical structure.

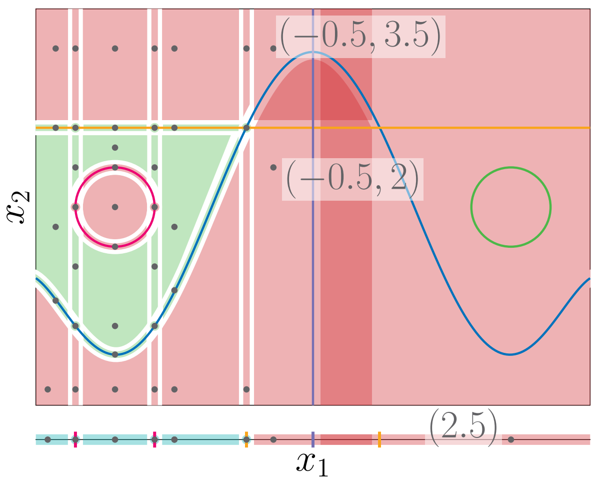
Definition 2.2 (Cylindrical Algebraic Decomposition [11]). *Let $i \in [n]$.*

- *An* (i-dimensional) cell *is a non-empty connected subset of $\mathbb{R}^i$.*
- *A* decomposition *of $\mathbb{R}^i$ is a finite set D_i of i-dimensional cells such that either $R = R'$ or $R \cap R' = \emptyset$ for all $R, R' \in D_i$, and $\cup_{R \in D_i} R = \mathbb{R}^i$.*
- *A set $R' \subseteq \mathbb{R}^i$ is* semi-algebraic *if it is the solution set of a quantifier-free formula.*
- *A decomposition D_i of $\mathbb{R}^i$ is* cylindrical *if either $i = 1$, or $i > 1$ and $D_{i-1} = \{R_{[i-1]} \mid R \in D_i\}$ is a cylindrical decomposition of $\mathbb{R}^{i-1}$.*
- *A* cylindrical algebraic decomposition (CAD) of $\mathbb{R}^i$ *is a cylindrical decomposition D of $\mathbb{R}^i$ whose cells $R \in D$ are semi-algebraic.*
- *Let $P \subseteq \mathbb{Q}[x_1, \ldots, x_n]$ and D be a CAD of $\mathbb{R}^n$. We call D* sign-invariant for P *if each $p \in P$ is sign-invariant on each cell $R \in D$.*
- *Let $R \subseteq \mathbb{R}^i$ be a cell and $f, g : R \to \mathbb{R}$ continuous functions with $f(s) < g(s)$ for all $s \in R$. We call $R \times \mathbb{R}$ the* cylinder over R, *$R \times [f, f]$ a* section over R, *and $R \times (f, g)$ a* sector over R. □

Unfortunately, the number of cells in a CAD that is sign-invariant for some set of polynomials may be doubly exponential in the number of their variables. This heavy complexity is partly rooted in the *projection phase* of the CAD method, in which polynomials of level i are projected to generate polynomials of level $i - 1$, iteratively for $i = n, \ldots, 2$. The effort for these projections depends on the degree and on the number of the polynomials, which might grow double-exponentially during this process. Thus, the costs of the projection plays a substantial role for the practical feasibility of the algorithm.



(a) Sign-invariant CAD from Example 2.2.



(b) Truth-invariant CAlC from Example 2.3, along with a sample point for each cell.

Figure 1: In the two-dimensional coordinate system, red cells evaluate the formula φ to *false*, and green cells evaluate φ to *true*. In the one-dimensional coordinate system, red cells evaluate φ to *false*, and in blue cells φ is satisfiable.

Example 2.2. *Consider the formula* $\varphi(x_1, x_2) = p_1 \leq 0 \wedge p_2 > 0 \wedge p_3 \geq 0 \wedge p_4 \leq 0 \wedge p_5 \leq 0$ *built using the following polynomials:*

$$p_1 = -0.006(x_1 - 2)(x_1 + 2)(x_1 - 3)(x_1 + 3)(x_1 - 4)(x_1 + 4) - x_2$$
$$p_2 = (x_1 + 2.5)^2 + (x_2 - 1.5)^2 - 0.25 \qquad p_4 = x_2 - 2.5$$
$$p_3 = (x_1 - 2.5)^2 + (x_2 - 1.5)^2 - 0.25 \qquad p_5 = x_1$$

The CAD method computes the projection polynomials $p_6 = \mathrm{disc}_{x_2}(p_2)$, $p_7 = \mathrm{disc}_{x_2}(p_3)$ *and* $p_8 = \mathrm{res}_{x_2}(p_1, p_4)$, *where disc and res are operators which we do not detail here. Figure 1a illustrates the variety of these polynomials, as well as the CAD of* $\mathbb{R}$ *and* $\mathbb{R}^2$.

Further, each cell is coloured according to the truth value (green for true and red for false) of φ *on that cell, which is determined by substituting a sample point (not depicted in the figure) from each cell into* φ.

If we aim to check whether $\exists x_1.\ \exists x_2.\ \varphi$ *holds, we only need to find one cell where* φ *evaluates to true. To check whether* $\exists x_1.\ \forall x_2.\ \varphi$ *holds, we would iterate through the one-dimensional CAD (e.g. finitely many values for* x_1*) and then check whether the respective cylinder is covered by cells where* φ *is true.* ⌟

A key observation is that the computed CAD is often finer than necessary for quantifier elimination. Some projection polynomials may be omitted, leading to a coarser CAD, because the real roots of the projection polynomials define the boundaries of the CAD cells. Now the idea of *exploration-guided* algorithms comes into play: we can iteratively guess sample points and generalize them to truth-invariant cells, and compute a coarser decomposition (or covering) of the state space by combining the individual generalizations. Such algorithms are *NuCAD* with a weaker notion of cylindricity, *CAlC* [1] which maintains cylindricity but relaxes decomposition to covering, and *NLSAT* [17] without an explicit cell structure but implicitly computes a something similar as CAlC (and is not directly extended for QE yet). In Section 5 we will compare NuCAD against CAlC, therefore we introduce it below.

Definition 2.3 (Cylindrical Algebraic Covering [1]). *Let* $i \in [n]$.

- *A* covering *of* $\mathbb{R}^i$ *is a finite set* D_i *of* i*-dimensional cells with* $\cup_{R \in D_i} R = \mathbb{R}^i$.
- *A covering* D_i *of* $\mathbb{R}^i$ *is* minimal *if there does not exist a covering* $D'_i \subsetneq D_i$ *of* $\mathbb{R}^i$.
- *A covering* D_i *of* $\mathbb{R}^i$ *is* cylindrical *if either* $i = 1$, *or* $i > 1$ *and* $D_{i-1} = \{R_{[i-1]} \mid R \in D_i\}$ *is a minimal cylindrical covering of* $\mathbb{R}^{i-1}$.

- *A* cylindrical algebraic covering (CAlC) of $\mathbb{R}^i$ *is a cylindrical covering D_i of $\mathbb{R}^i$ whose cells $R \in D_i$ are semi-algebraic.*
- *Let φ be a formula with free variables $x_1, \dots, x_n$. A CAlC D_n of $\mathbb{R}^n$ is* truth-invariant for φ *if φ is truth-invariant on each cell $R \in D_n$.* □

Example 2.3. *A truth-invariant CAlC for the formula φ from Example 2.2 is depicted in Figure 1b. We emphasize that due to the cylindrical arrangement, we can reason about the formula and its quantifiers similarly as with a CAD.* ⌟

2.2. Single Cylindrical Cells

A key concept for exploration-guided algorithms is the *single cell construction* [6, 9, 22], which takes a set of polynomials and a sample point as input, and computes the description of a cell that contains the sample point, such that each input polynomial is sign-invariant over the cell. These cells have the following property, which allows to describe them explicitly:

Definition 2.4 (Local Cylindricity [9]). *Let $i \in [n]$. A set $R \subseteq \mathbb{R}^i$ is* locally cylindrical *if either $i = 1$, or $i > 1$ and R is a section or sector over $R_{[i-1]}$ which itself is locally cylindrical.* □

We can thus describe locally cylindrical cells by bounds on x_1, bounds on x_2 that depend on x_1, bounds on x_3 that depend on x_1 and x_2, and so on. These bounds are described as varieties of some projection polynomials, e.g. each bound is some root of a polynomial. We formalize these notions using symbolic intervals.

Definition 2.5 (Indexed Root Expression, Symbolic Interval). *An* indexed root expression (of level i) *is a partial function $\mathrm{root}_{x_i}^{p,j} : \mathbb{R}^{i-1} \hookrightarrow \mathbb{R}$ for some $i, j \in \mathbb{N}_{>0}$ and $p \in \mathbb{Q}[x_1, \dots, x_i]$ with $\mathrm{level}(p) = i$, such that for each $s \in \mathbb{R}^{i-1}$, $\mathrm{root}_{x_i}^{p,j}(s)$ is the j-th real root of the univariate polynomial $p(s) \in \mathbb{Q}[x_i]$ if it exists, and it is undefined (undef) otherwise.*

A symbolic interval $\mathtt{I}$ of level i *has the form $\mathtt{I} = (l, u)$, $\mathtt{I} = [l, u]$, $\mathtt{I} = [l, u)$, or $\mathtt{I} = (l, u]$ where l is either an indexed root expression of level i or $-\infty$, and u is either an indexed root expression of level i or ∞. The polynomials $\mathtt{I}.l$ and $\mathtt{I}.u$ are the* defining polynomials of $\mathtt{I}$ *(if they exist).* □

We use the cross product notation also for symbolic intervals.
Algorithm 1 computes such a single cell using the levelwise algorithm from [22]. It uses Algorithm 2 to determine the symbolic interval $\mathtt{I}$ on a given level: by isolating the real roots of the polynomials, it determines which polynomial bounds the cell on that level. It then uses Algorithm 3 to compute projection polynomials that characterize an underlying cell where the bounds described by $\mathtt{I}$ remain valid. For $P \subseteq \mathbb{Q}[x_1, \dots, x_i]$ and $s \in \mathbb{R}^i$, let $C(P, s) \subseteq \mathbb{R}^i$ denote the inclusion-maximal P-sign-invariant cell that contains s. Then due to its local cylindricity, the resulting cell $\mathtt{I}_1 \times \ldots \times \mathtt{I}_n$ will be a subset of $C(P, s)$. For the details of Algorithm 3, we refer to [22, Algorithm 2] and [21, Algorithm 7].

Example 2.4. *Consider the polynomials $p_1 = 0.5x_1 + 0.5 - x_2$, $p_2 = x_1^2 + x_2^2 - 1$, $p_3 = 0.5x_1 - 0.5 - x_2$, and the point $s = (0.125, -0.75)$.*

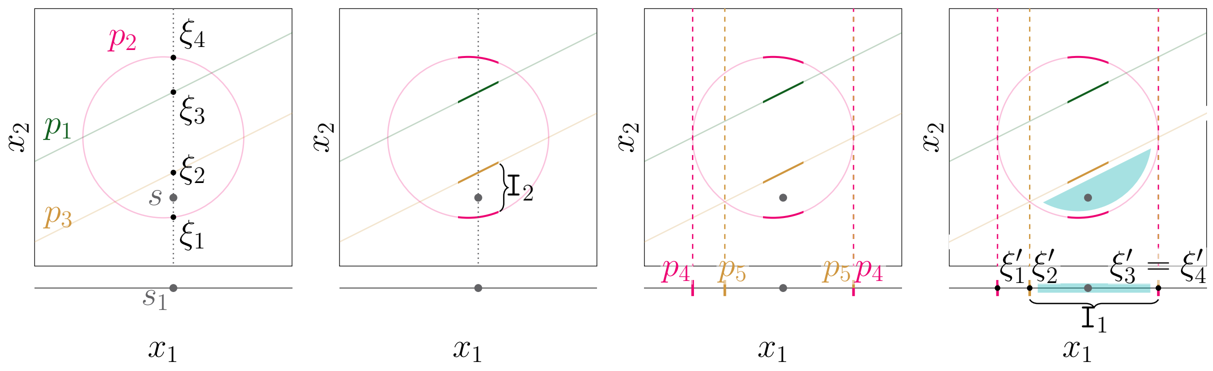
We consider the first coordinate $s_1 = 0.125$ of s, evaluate the polynomials at this value for x_1, and isolate the real roots of the resulting univariate polynomials to find $\xi_1 = \mathrm{root}_{x_2}^{p_2,1}$, $\xi_2 = \mathrm{root}_{x_2}^{p_3,1}$, $\xi_3 = \mathrm{root}_{x_2}^{p_1,1}$, and $\xi_4 = \mathrm{root}_{x_2}^{p_2,2}$. We order these, along with the second coordinate of s, as in Figure 2a. The symbolic

Algorithm 1: `construct_cell`(P,s).

```
Input  : $P = P' \subseteq \mathbb{Q}[x_1, \dots, x_n]$, $s \in \mathbb{R}^n$.
Output : Symbolic intervals $\mathtt{I}_1, \dots, \mathtt{I}_n$ of levels $1, \dots, n$ respectively such that $\mathtt{I}_1 \times \ldots \times \mathtt{I}_n \subseteq C(P, s)$.
foreach $i = n, \dots, 1$ do
    $\mathtt{I}_i$ := choose_interval($P'$,$s_{[i]}$)                    // Algorithm 2
    $P'$ := compute_cell_projection($P'$,$s_{[i]}$,$\mathtt{I}_i$)        // Algorithm 3
return $\mathtt{I}_1, \dots, \mathtt{I}_n$
```

Algorithm 2: `choose_interval(`P,s`)`.

```
Input  : $P \subseteq \mathbb{Q}[x_1, \ldots, x_i]$, $s \in \mathbb{R}^i$ for an $i \in [n]$.
Output : Symbolic interval $\mathtt{I}$ of level $i$ such that $s_{[i-1]} \times \mathtt{I} = s_{[i-1]} \times \{r \in \mathbb{R} \mid (s_{[i-1]}, r) \in C(P, s)\}$.
1  $\Xi := \cup_{p \in P}$ realRoots($p(s_{i-1})$)
2  if $\Xi = \emptyset$ then return $(-\infty, \infty)$
3  otherwise assume $\Xi = \{\xi_1, \ldots, \xi_k\}$ such that $\xi_1 < \ldots < \xi_k$
4  if $s_i \in (-\infty, \xi_1)$ then
5      return $(-\infty, \mathrm{root}^{p,j}_{x_i})$ for $p \in P$, $j \in \mathbb{N}_{>0}$ such that $\xi_1 = \mathrm{root}^{p,j}_{x_i}(s_{[i-1]})$
6  else if $s_i \in (\xi_\ell, \xi_{\ell+1})$ then
7      return $(\mathrm{root}^{p,j}_{x_i}, \mathrm{root}^{p',j'}_{x_i})$ for $p, p' \in P$, $j, j' \in \mathbb{N}_{>0}$ such that $\xi_\ell = \mathrm{root}^{p,j}_{x_i}(s_{[i-1]})$ and
        $\xi_{\ell+1} = \mathrm{root}^{p',j'}_{x_i}(s_{[i-1]})$
8  else if $s_i = \xi_\ell$ then
9      return $[\mathrm{root}^{p,j}_{x_i}, \mathrm{root}^{p,j}_{x_i}]$ for $p \in P$, $j \in \mathbb{N}_{>0}$ such that $\xi_\ell = \mathrm{root}^{p,j}_{x_i}(s_{[i-1]})$
10 else if $s_i \in (\xi_k, \infty)$ then
11     return $(\mathrm{root}^{p,j}_{x_i}, \infty)$ for $p \in P$, $j \in \mathbb{N}_{>0}$ such that $\xi_k = \mathrm{root}^{p,j}_{x_i}(s_{[i-1]})$
```



(a) The sample point depicted on the one- and two-dimensional coordinate systems.

(b) The real roots witness root functions of the polynomials.

(c) Zeros of projection polynomials define the symbolic interval bound.

(d) On the level below, we iterate the process.

Figure 2: Levelwise construction of the single cell from Example 2.4.

interval containing s is denoted $\mathtt{I}_2 = (\xi_1, \xi_2)$ in Figure 2b. Thus local to s_1, the cell we want is bounded from below by p_2 and from above by p_3.

As we generalize from s we must ensure that the domains of ξ_1 and ξ_2 remain well-defined, no root function crosses the symbolic interval $\mathtt{I}_2$, and no additional roots pop up within $\mathtt{I}_2$. To achieve this, we compute the projection polynomials p_4 and p_5 (details omitted here). Figure 2c shows how the zeros of the projection map onto the geometric features. Analogously to x_2, we isolate the real roots $\xi_1' = \mathrm{root}^{p_4,1}_{x_1}, \xi_2' = \mathrm{root}^{p_5,1}_{x_1}, \xi_3' = \mathrm{root}^{p_4,2}_{x_1}, \xi_4' = \mathrm{root}^{p_5,2}_{x_1}$ in x_1. We determine the interval $\mathtt{I}_1 = (\xi_2', \xi_3')$ around s_1 for x_1 and thus generate the cell in Figure 2d. ⌟

3. Non-uniform Cylindrical Algebraic Decomposition

The NuCAD algorithm builds — similarly to NLSAT and CAlC — on top of the single cell construction. It computes a *decomposition* of $\mathbb{R}^n$, whose cells, however, are not *globally* cylindrically arranged. It can

Algorithm 3: `compute_cell_projection(`P,s,$\mathtt{I}$`)`.

```
Input  : $P \subseteq \mathbb{Q}[x_1, \ldots, x_i]$, $s \in \mathbb{R}^i$, symbolic interval $\mathtt{I}$ of level $i$ for an $i \in [n]$.
Output : $P' \subseteq \mathbb{Q}[x_1, \ldots, x_{i-1}]$ such that $C(P', s_{[i-1]}) \times \mathtt{I} \subseteq C(P, s)$.
```

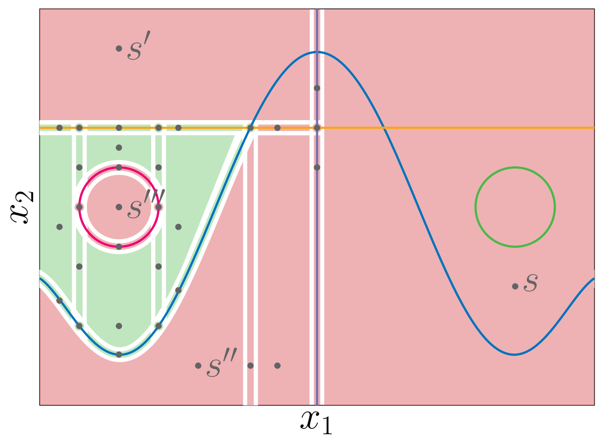


Figure 3: Truth-invariant non-uniform CAD, with a sample point for each cell.

be seen as a cylindrical decomposition, where some cells are further refined into a *local* cylindrical decomposition.

Definition 3.1 (Non-uniform CAD [7]). *Let $i \in [n]$, and $R \subseteq \mathbb{R}^i$ an algebraic cell. A* non-uniform cylindrical algebraic decomposition (NuCAD) of *R is an algebraic decomposition D of R that consists of locally cylindrical cells such that there exists a partition $P_1, \ldots, P_k$ of D (that is $P_1 \cup \ldots \cup P_k = D$ is a disjoint union) such that $\{\cup_{C \in P_j} C \mid j \in [k]\}$ is cylindrical, and for each $j \in [k]$ the set P_j consists either of a single element or is a NuCAD of $\cup_{C \in P_j} C$.* □

Example 3.1. *We consider the formula from Example 2.2 again. A truth-invariant NuCAD for the input formula is depicted in Figure 3.* ⌟

We can represent a NuCAD by the following data structure:

Definition 3.2 (NuCAD Data Structure). *A* NuCAD structure of level *$i \in [n]$* w.r.t. global level *$i' \in [i..n]$ is a finite sequence of pairs $(\mathtt{I}_1, \mathtt{T}_1), \ldots, (\mathtt{I}_k, \mathtt{T}_k)$ with $k \in \mathbb{N}_{>0}$ and for all $j \in [k]$, $\mathtt{I}_j$ is a symbolic interval of level i and $\mathtt{T}_j$ is either true, false, a NuCAD structure of level $i+1 \leq i'$ w.r.t. global level i', or a NuCAD structure of level 1 w.r.t. global level i'.* □

We omit a formal semantics due to the space limit. Intuitively, each subtree of some node either describes a decomposition of the cylinder above the cell described by the intervals on the path to the given node (as long as the levels are increasing), or describes a decomposition of that cell (if the level is reset to 1). We represent the NuCAD recursively to retain information about the arrangement of the cells, similar to [7, Definition 2]. The technicalities of this definition are not crucial for the NuCAD algorithm. We thus refrain from going into more detail.

Example 3.2. *The NuCAD depicted in Figure 3 is represented by the tree partially depicted in Figure 4. A path from the root of length 2 describes a cell of the decomposition, and the following subtree the decomposition of that cell, e.g., $\mathtt{T}_0$ is the tree decomposing $\mathbb{R}^2$, $\mathtt{T}_1$ is the decomposition of the first cell $(-\infty, \mathrm{root}_{x_1}^{p_5,1}) \times (-\infty, \infty)$ of $\mathtt{T}_1$. Figure 5 displays the corresponding decompositions.* ⌟

3.1. A Complete Algorithm for Computing a NuCAD

We present a version of the algorithm from [9, Algorithm 2] for NuCAD computation, which we adapted to be *complete* (i.e. not restricted to open cells only). To handle the Boolean structure of the formula, we make use of Algorithm 4 for computing implicants as in Definition 2.1. For details, we refer to [21, Section 6].

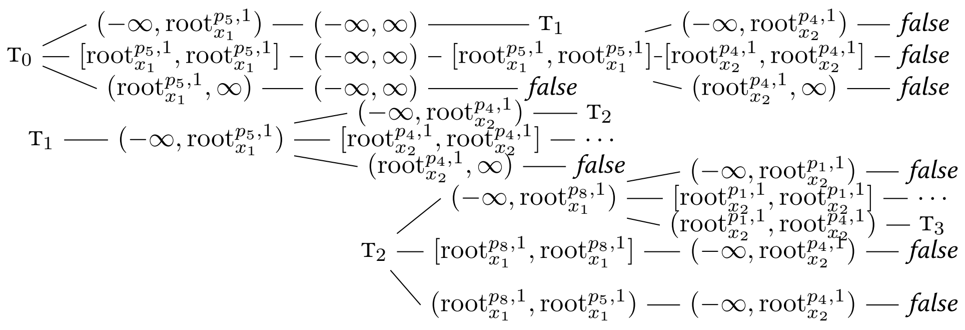
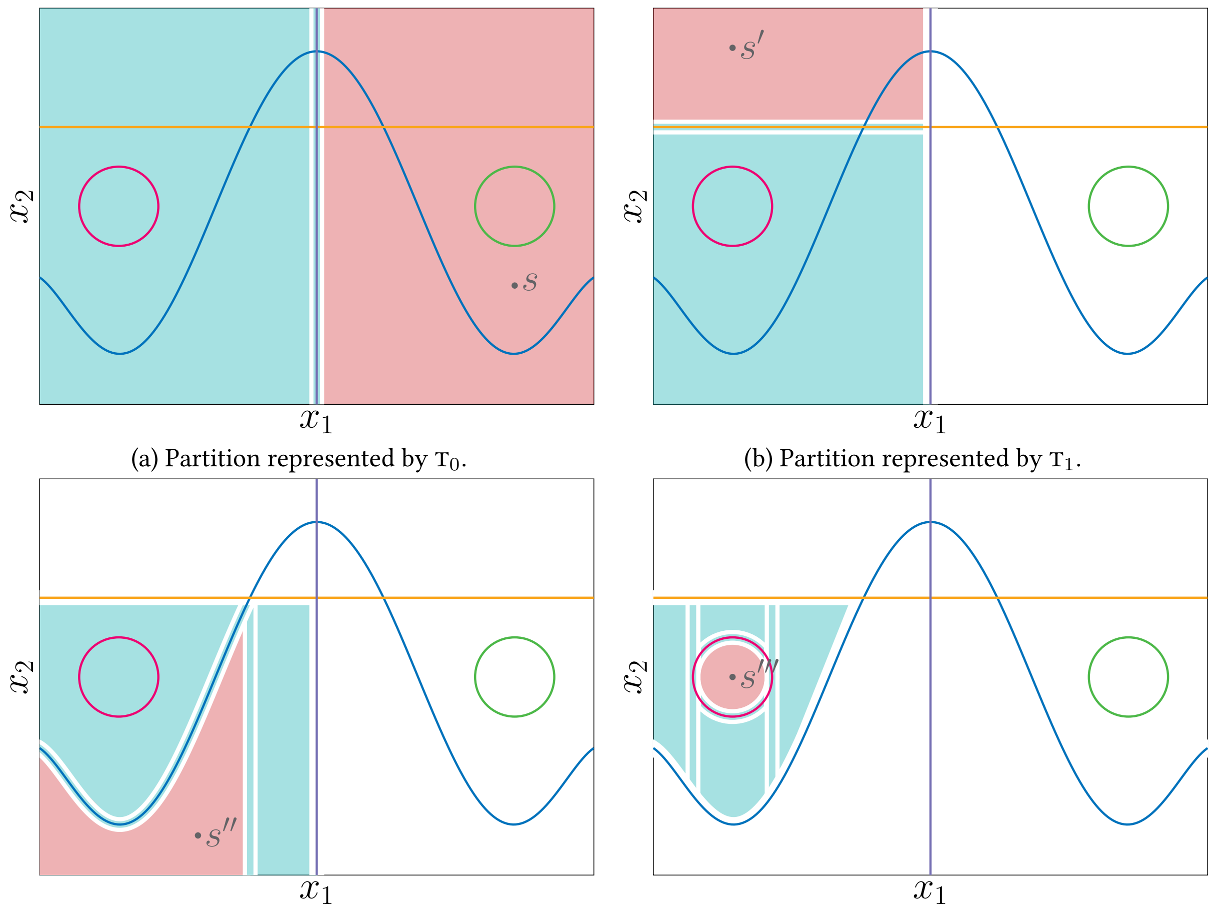

$\mathtt{T}_0$
$(-\infty, \mathrm{root}_{x_1}^{p_5,1})$ — $(-\infty,\infty)$ — $\mathtt{T}_1$
$[\mathrm{root}_{x_1}^{p_5,1}, \mathrm{root}_{x_1}^{p_5,1}]$ – $(-\infty,\infty)$ – $[\mathrm{root}_{x_1}^{p_5,1}, \mathrm{root}_{x_1}^{p_5,1}]$-$[\mathrm{root}_{x_2}^{p_4,1}, \mathrm{root}_{x_2}^{p_4,1}]$ – *false*
$(-\infty, \mathrm{root}_{x_2}^{p_4,1})$ — *false*
$(\mathrm{root}_{x_2}^{p_4,1}, \infty)$ — *false*
$(\mathrm{root}_{x_1}^{p_5,1}, \infty)$ — $(-\infty,\infty)$ — *false*
$\mathtt{T}_1$ — $(-\infty, \mathrm{root}_{x_1}^{p_5,1})$
$(-\infty, \mathrm{root}_{x_2}^{p_4,1})$ — $\mathtt{T}_2$
$[\mathrm{root}_{x_2}^{p_4,1}, \mathrm{root}_{x_2}^{p_4,1}]$ — ...
$(\mathrm{root}_{x_2}^{p_4,1}, \infty)$ — *false*
$\mathtt{T}_2$
$(-\infty, \mathrm{root}_{x_1}^{p_8,1})$
$(-\infty, \mathrm{root}_{x_2}^{p_1,1})$ — *false*
$[\mathrm{root}_{x_2}^{p_1,1}, \mathrm{root}_{x_2}^{p_1,1}]$ — ...
$(\mathrm{root}_{x_2}^{p_1,1}, \mathrm{root}_{x_2}^{p_4,1})$ — $\mathtt{T}_3$
$[\mathrm{root}_{x_1}^{p_8,1}, \mathrm{root}_{x_1}^{p_8,1}]$ — $(-\infty, \mathrm{root}_{x_2}^{p_4,1})$ — *false*
$(\mathrm{root}_{x_1}^{p_8,1}, \mathrm{root}_{x_1}^{p_5,1})$ — $(-\infty, \mathrm{root}_{x_2}^{p_4,1})$ — *false*

Figure 4: NuCAD data structure that represents the NuCAD in Figure 3.



(a) Partition represented by $\mathtt{T}_0$.

(b) Partition represented by $\mathtt{T}_1$.

(c) Partition represented by $\mathtt{T}_2$.

(d) Partition represented by $\mathtt{T}_3$.

Figure 5: Some partitions of cells in the NuCAD from Figures 3 and 4.

Let us assume that we generate a NuCAD of some cell $R = \mathtt{I}_1 \times \ldots \times \mathtt{I}_n \subseteq \mathbb{R}^n$ for a quantifier-free formula φ using Algorithm 5; for now, ignore the highlighted (but consider the underlined) bits in the algorithm. We start by picking a sample point contained in R in Algorithm 5, and in Algorithm 4, we obtain an implicant for φ w.r.t. s. In Algorithm 5, we apply the single cell construction algorithm on the obtained implicit cell. To compute a cell that is a subset of $R \subseteq \mathbb{R}^n$, we insert the defining polynomials of $\mathtt{I}_1, \ldots, \mathtt{I}_n$ to P in Algorithm 5 before constructing the cell.

The obtained cell $R' = \mathtt{I}'_1 \times \ldots \times \mathtt{I}'_n$ is thus a subset of R, and φ is truth-invariant on R'. We insert the cell into the NuCAD data structure $\mathtt{T}$ in Algorithm 5, such that — when seen as a tree — there is a path $\mathtt{I}'_1, \ldots, \mathtt{I}'_n$ from the root node leading to $\mathtt{t}$. What remains is to explore the remaining parts $R \setminus R'$. We do so by splitting R using R' into multiple cylindrical cells in Algorithm 5: we observe that a point is outside the cell if it violates one bound. We thus iterate through each level i and encode that the first $i-1$ levels are in R'; on the i-th level, we leave R' (we are either below or above the cell, if possible) but stay in R; and on the levels starting from $i+1$, we remain in R, but do not make assumptions about the bounds on R', as we already left R' on the i-th level.

We end up with a set Q of unexplored cells which are locally cylindrical, i.e. they are described by symbolic intervals for each variable. We call the NuCAD algorithm on each cell in Q in Algorithm 5 to explore the remaining parts of R.

Example 3.3. *Consider the formula $\varphi(x_1, x_2) = p_1 \leq 0 \wedge p_2 > 0 \wedge p_3 \geq 0 \wedge p_4 \leq 0 \wedge p_5 \leq 0$ and the polynomials from Example 2.2 again. We give an exemplary run of Algorithm 5 that computes the NuCAD as depicted in Figures 3 and 4.*

`nucad_full`(φ, $((-\infty,\infty),(-\infty,\infty))$) *We choose s as sample point. As φ evaluates to false at s, we obtain the implicant $p_5 > 0$, and thus obtain $(\{p_5\}, s)$ as implicit cell. By single cell construction, we obtain the cell described by $\mathtt{I}_1' \times \mathtt{I}_2' = (\mathrm{root}_{x_1}^{p_5,1}, \infty) \times (-\infty,\infty)$. Accordingly, we split $(-\infty,\infty) \times (-\infty,\infty)$ into $(-\infty, \mathrm{root}_{x_1}^{p_5,1}) \times (-\infty,\infty)$, $[\mathrm{root}_{x_1}^{p_5,1}, \mathrm{root}_{x_1}^{p_5,1}] \times (-\infty,\infty)$, and $\mathtt{I}_1' \times \mathtt{I}_2'$, as depicted in Figure 5a.*

> `nucad_full`(φ, $((-\infty,\mathrm{root}_{x_1}^{p_5,1}),(-\infty,\infty))$) *We choose the sample point s', and obtain the implicit cell $(\{p_4\}, s')$ where φ is truth-invariant. As we aim to compute a truth-invariant cell that is a subset of $(-\infty, \mathrm{root}_{x_1}^{p_5,1}) \times (-\infty,\infty)$, we run single cell construction on $(\{p_4, p_5\}, s')$. We obtain $\mathtt{I}_1' \times \mathtt{I}_2' = (-\infty, \mathrm{root}_{x_1}^{p_5,1}) \times (\mathrm{root}_{x_2}^{p_4,1}, \infty)$, and split into $(-\infty, \mathrm{root}_{x_1}^{p_5,1}) \times (-\infty, \mathrm{root}_{x_2}^{p_4,1})$, $(-\infty, \mathrm{root}_{x_1}^{p_5,1}) \times [\mathrm{root}_{x_2}^{p_4,1}, \mathrm{root}_{x_2}^{p_4,1}]$, and $\mathtt{I}_1' \times \mathtt{I}_2'$, as depicted in Figure 5b.*
>
> > `nucad_full`(φ, $((-\infty,\mathrm{root}_{x_1}^{p_5,1}),(-\infty,\mathrm{root}_{x_2}^{p_4,1}))$) *We choose s'', and obtain $(\{p_1\}, s'')$ from the implicant. We run single cell construction on $(\{p_1, p_4, p_5\}, s'')$ and obtain $(-\infty, \mathrm{root}_{x_1}^{p_8,1}) \times (-\infty, \mathrm{root}_{x_2}^{p_1,1})$ (where p_8 is the resultant of p_1 and p_4), and split the input cell into the cells as indicated in Figure 5c and the following recursive calls:*
> >
> > `nucad_full`(…, $([\mathrm{root}_{x_1}^{p_8,1}, \mathrm{root}_{x_1}^{p_8,1}], (-\infty, \mathrm{root}_{x_2}^{p_4,1}))$) …
> >
> > `nucad_full`(…, $((\mathrm{root}_{x_1}^{p_8,1}, \mathrm{root}_{x_1}^{p_5,1}), (-\infty, \mathrm{root}_{x_2}^{p_4,1}))$) …
> >
> > `nucad_full`(…, $((-\infty, \mathrm{root}_{x_1}^{p_8,1}), [\mathrm{root}_{x_2}^{p_1,1}, \mathrm{root}_{x_2}^{p_1,1}])$) …
> >
> > `nucad_full`(…, $((-\infty, \mathrm{root}_{x_1}^{p_8,1}), (\mathrm{root}_{x_2}^{p_1,1}, \mathrm{root}_{x_2}^{p_4,1}))$) *We choose s'' and obtain $(\{p_2\}, s''')$ from the implicant. We run single cell construction on $(\{p_1, p_2, p_4\}, s''')$ and obtain $(\mathrm{root}_{x_1}^{p_6,1}, \mathrm{root}_{x_1}^{p_6,2}) \times (\mathrm{root}_{x_2}^{p_2,1}, \mathrm{root}_{x_2}^{p_2,2})$ (where p_6 is the resultant of p_2), and split the input cell into the cells as indicated in Figure 5d and computed by some recursive calls (omitted due to limited space).*
>
> `nucad_full`(φ, $((-\infty,\mathrm{root}_{x_1}^{p_5,1}),[\mathrm{root}_{x_2}^{p_4,1}, \mathrm{root}_{x_2}^{p_4,1}])$) …

`nucad_full`(φ, $([\mathrm{root}_{x_1}^{p_5,1}, \mathrm{root}_{x_1}^{p_5,1}],(-\infty,\infty))$) … ⌟

4. Quantifier Alternation in NuCAD

A NuCAD as computed by the algorithm presented in the previous section is not eligible for reasoning about quantifier alternations, as it does not have the cylindrical structure. An extension for quantifier elimination is given in [8], where a NuCAD of $\mathbb{R}^n$ is refined such that reasoning about quantifiers is possible.

Algorithm 4: `choose_enclosing_cell`(φ, s).

```
Input  :Formula $\varphi$, $s \in \mathbb{R}^i$ for an $i \in [n]$ s.t. $\varphi[s] \equiv$ false or $\varphi[s] \equiv$ true.
Output :$P \subseteq \mathbb{Q}[x_1, \ldots, x_i]$, t $\in \{$true, false$\}$ s.t. $\varphi(s') \equiv$ t for all $s' \in C(P, s)$.
```

Algorithm 5: `nucad_full`(φ, s, $(\mathtt{I}_1, \ldots, \mathtt{I}_n)$ $(\mathtt{I}_{i+1}, \ldots, \mathtt{I}_{i'})$).

```
   Input  : Formula $\varphi$, $s \in \mathbb{R}^i$ for an $i \in [0..n]$,
            symbolic intervals $(\mathtt{I}_1, \ldots, \mathtt{I}_n)$ $(\mathtt{I}_{i+1}, \ldots, \mathtt{I}_{i'})$ for an $i' \in [i+1..n]$.
   Output : $P \subseteq \mathbb{Q}[x_1, \ldots, x_i]$, and NuCAD data structure $\mathtt{T}$ representing a truth-invariant NuCAD of
            $\mathtt{I}_1 \times \ldots \times \mathtt{I}_n$ $C(P,s) \times \mathtt{I}_{i+1} \times \ldots \times \mathtt{I}_{i'}$ *over* $C(P,s)$.
1  choose $r \in \mathbb{R}^n$ s.t. $r \in \mathtt{I}_1 \times \ldots \times \mathtt{I}_n$ $r \in \mathbb{R}^{i'}$ s.t. $r \in s \times \mathtt{I}_{i+1} \times \ldots \times \mathtt{I}_{i'}$
2  $P, \mathtt{t}$ := choose_enclosing_cell($\varphi, r$) nucad_recurse($\varphi, r$)
3  $P := P \cup \{\mathtt{I}_j.l.p \mid j \in [1..n]\ [i+1..i'], \mathtt{I}_j.l \neq -\infty\} \cup$
        $\{\mathtt{I}_j.u.p \mid j \in [1..n]\ [i+1..i'], \mathtt{I}_j.u \neq \infty\}$
4  foreach $j = n, \ldots, 1$ $i', \ldots, i+1$ do
5      $\mathtt{I}'_j$ := choose_interval($P$,$r_{[j]}$)                       // Algorithm 2
6      $P$ := compute_cell_projection($P$,$r_{[j]}$,$\mathtt{I}'_j$)            // Algorithm 3
7  $\mathtt{T} := \emptyset$
8  insert $\mathtt{I}'_1, \ldots, \mathtt{I}'_n$ $\mathtt{I}'_{i+1}, \ldots, \mathtt{I}'_{i'}$ into $\mathtt{T}$ leading to $\mathtt{t}$
9  $Q := \emptyset$
10 foreach $j = 1, \ldots, n$ $i+1, \ldots, i'$ do
11     if $\mathtt{I}'_j.l \neq \mathtt{I}_j.l$ then
12         $Q := Q \cup \{(\mathtt{I}'_1\ \mathtt{I}'_{i+1}, \ldots, \mathtt{I}'_{j-1}, (\mathtt{I}_j.l, \mathtt{I}'_j.l), \mathtt{I}_{j+1}, \ldots, \mathtt{I}_n\ \mathtt{I}_{i'})\}$
13         if $\mathtt{I}'_j$ is a sector then
14             $Q := Q \cup \{(\mathtt{I}'_1\ \mathtt{I}'_{i+1}, \ldots, \mathtt{I}'_{j-1}, [\mathtt{I}'_j.l, \mathtt{I}'_j.l], \mathtt{I}_{j+1}, \ldots, \mathtt{I}_n\ \mathtt{I}_{i'})\}$
15     if $\mathtt{I}'_j.u \neq \mathtt{I}_j.u$ then
16         $Q := Q \cup \{(\mathtt{I}'_1\ \mathtt{I}'_{i+1}, \ldots, \mathtt{I}'_{j-1}, (\mathtt{I}'_j.u, \mathtt{I}_j.u), \mathtt{I}_{j+1}, \ldots, \mathtt{I}_n\ \mathtt{I}_{i'})\}$
17         if $\mathtt{I}'_j$ is a sector then
18             $Q := Q \cup \{(\mathtt{I}'_1\ \mathtt{I}'_{i+1}, \ldots, \mathtt{I}'_{j-1}, [\mathtt{I}'_j.u, \mathtt{I}'_j.u], \mathtt{I}_{j+1}, \ldots, \mathtt{I}_n\ \mathtt{I}_{i'})\}$
19 foreach $(\mathtt{I}'_1, \ldots, \mathtt{I}'_n)$ $(\mathtt{I}'_{i+1}, \ldots, \mathtt{I}'_{i'}) \in Q$ do
20     $P'$, $\mathtt{T}'$ := nucad_full($\varphi$, $s$, $(\mathtt{I}'_1, \ldots, \mathtt{I}'_n)$ $(\mathtt{I}'_{i+1}, \ldots, \mathtt{I}'_{i'})$)
21     $P := P \cup P'$
22     insert $\mathtt{I}'_1, \ldots, \mathtt{I}'_n$ $\mathtt{I}'_{i+1}, \ldots, \mathtt{I}'_{i'}$ into $\mathtt{T}$ leading to $\mathtt{T}'$
23 return $P$, $\mathtt{T}$
```

Algorithm 6: `nucad_recurse`(φ, s).

```
   Input  : Formula $\varphi$ in prenex normal form, $s \in \mathbb{R}^i$ for an $i \in [0..n]$ s.t. $\varphi$ contains at most $i$ free variables.
   Output : $P \subseteq \mathbb{Q}[x_1, \ldots, x_i]$, $\mathtt{t} \in \{true, false\}$ s.t. $\varphi(s') \equiv \mathtt{t}$ for all $s' \in C(P,s)$.
1  if $\varphi[s] \equiv false \lor \varphi[s] \equiv true$ then
2      return choose_enclosing_cell($\varphi$, $s$)                                 // Algorithm 4
3  else it holds $i < n$
4      if $\varphi = \exists x_{i+1} \ldots \exists x_{i'}.\ \psi$ then
5          return nucad_quantifier($\exists, \psi, s, i', (-\infty,\infty), \ldots, (-\infty,\infty)$)      // A 7
6      else if $\varphi = \forall x_{i+1} \ldots \forall x_{i'}.\ \psi$ then
7          return nucad_quantifier($\forall, \psi, s, i', (-\infty,\infty), \ldots, (-\infty,\infty)$)      // A 7
```

Borrowing ideas from CAlC, we present an adaptation which directly computes a NuCAD adapted to the input's quantifier structure, making the refinement step obsolete. For doing so, we define a NuCAD over a cell, which allows to generalize the notion of cylindricity from single levels to blocks of levels.

Definition 4.1. *Let $i \in [0..n]$, $i' \in [i+1..n]$, and $R \subseteq \mathbb{R}^{i'}$ be an algebraic cell. A* NuCAD of R over $R_{[i]}$ *is a NuCAD D of R s.t. $R'_{[i]} = R_{[i]}$ for all $R' \in D$.* □

If we call Algorithm 5 — now we consider the highlighted and ignore the underlined parts — with a sample point $s \in \mathbb{R}^i$ and $i' < n$, it will compute a NuCAD over some cell around s of $\mathbb{R}^{i'}$. For doing so, it fixes the first i levels of the guessed sample point in Algorithm 5, applies single cell construction only from level i' to level $i+1$ in Algorithm 5, applies the splitting accordingly in Algorithm 5, and — most importantly for the correctness — collects the polynomials that characterize the NuCAD. This set consists of the polynomials of level i and below that are left over from the single cell construction, and the projections obtained from the recursive calls to NuCAD (see Algorithm 5). Further, as we do not

Algorithm 7: nucad_quantifier(Q, φ, s, $(\mathtt{I}_{i+1}, \dots, \mathtt{I}_{i'})$).

```
Input  : $Q \in \{\exists, \forall\}$, formula $\varphi$, $s \in \mathbb{R}^i$ for an $i \in [0..n]$,
         symbolic intervals $(\mathtt{I}_{i+1}, \dots, \mathtt{I}_{i'})$ for an $i' \in [i+1..n]$.
Output : $P \subseteq \mathbb{Q}[x_1, \dots, x_i]$, $\mathtt{t} \in \{true, false\}$ s.t. $\varphi(s') \equiv \mathtt{t}$ for all $s' \in C(P, s)$.
1  choose $r \in \mathbb{R}^{i'}$ s.t. $r \in s \times \mathtt{I}_{i+1} \times \dots \times \mathtt{I}_{i'}$
2  $P, \mathtt{t}$ := nucad_recurse($\varphi$, $r$)                                  // Algorithm 6
3  if $(Q = \exists \wedge \mathtt{t} = true) \vee (Q = \forall \wedge \mathtt{t} = false)$ then
4      foreach $j = i', \dots, i{+}1$ do
5          $\mathtt{I}'_j$ := choose_interval($P$,$r_{[j]}$)                        // Algorithm 2
6          $P$ := compute_cell_projection($P$,$r_{[j]}$,$\mathtt{I}'_j$)              // Algorithm 3
7      return $P$, $\mathtt{t}$
8  $P := P \cup \{\mathtt{I}_j.l.p \mid j \in [i{+}1..i'], \mathtt{I}_j.l \neq -\infty\} \cup \{\mathtt{I}_j.u.p \mid j \in [i{+}1..i'], \mathtt{I}_j.u \neq \infty\}$
9  foreach $j = i', \dots, i{+}1$ do
10     $\mathtt{I}'_j$ := choose_interval($P$,$r_{[j]}$)                            // Algorithm 2
11     $P$ := compute_cell_projection($P$,$r_{[j]}$,$\mathtt{I}'_j$)                  // Algorithm 3
12 $Q := \emptyset$
13 for-loop in Lines 10-18 of Algorithm 5
14 foreach $(\mathtt{I}'_{i+1}, \dots, \mathtt{I}'_{i'}) \in Q$ do
15     $P', \mathtt{t}$ := nucad_quantifier($Q$, $\varphi$, $s$, $(\mathtt{I}'_{i+1}, \dots, \mathtt{I}'_{i'})$)
16     if $(Q = \exists \wedge \mathtt{t} = true) \vee (Q = \forall \wedge \mathtt{t} = false)$ then
17         return $P'$, $\mathtt{t}$
18     $P := P \cup P'$
19 if $Q = \exists$ then $\mathtt{t} := false$ else $\mathtt{t} := true$
20 return $P$, $\mathtt{t}$
```

compute a NuCAD of $\mathbb{R}^n$ but of $\mathbb{R}^{i'}$, the call to Algorithm 6 in Algorithm 5 gets important, as it obtains a truth-invariant cell for φ even if s is only a partial sample point.

Algorithm 6 decides whether to call Algorithm 4 or the NuCAD algorithm (we could replace `nucad_quantifier(...)` by `nucad_full(...)`). For handling formulas with quantifiers, it "structures" the NuCAD such that we construct a NuCAD for each quantifier block separately.

Although we could use Algorithm 5 for quantifier blocks, we only use it for describing the parameter space (by choosing i' as the highest index of a free variable in the input formula); for quantifier blocks, similarly to the CAlC algorithm, we can terminate the NuCAD construction earlier, depending on the quantifier: if the given variables are existentially quantified, we can stop when we find a cell where the input formula is *true*, and return a projection of that cell (instead of the NuCAD). Analogously, if the given variables are universally quantified, we stop when we find a cell where the input formula is *false*. Algorithm 7 implements this idea and differs from Algorithm 5 only in two places: if a call to Algorithm 6 yields a desired cell, we call single cell construction and return the result in Algorithm 7. If a recursive call to itself yields such a cell, we return that cell in Algorithm 7, omitting the intermediate results from the NuCAD computation.

5. Experimental Evaluation

We implemented the NuCAD algorithm in SMT-RAT [12], which also provides a CAlC implementation. Both implementations are sequential and share large portions of code (e.g. for Boolean reasoning as described in [21], CAD projection as described in [22], the variable ordering from [24], an adaption for the projection in [20] described in Section A.1, and post-processing as described in Section A.2), increasing their comparability. We run our experiments on Intel®Xeon®8468 Sapphire CPUs with 2.1 GHz per core with a time limit of 60 seconds and memory limit of 4GB per formula. For the evaluation, we use all instances from the *QF_NRA* and *NRA* benchmark sets from *SMT-LIB* [25] for quantifier-free and quantified decision problems, and a collection of quantifier elimination (*QE*) problems from Wilson [28]. The code, raw results and evaluation scripts are available at https://doi.org/10.5281/zenodo.15302831. We verified all results of our implementations using QEPCAD B/Tarski for quantified formulas, and

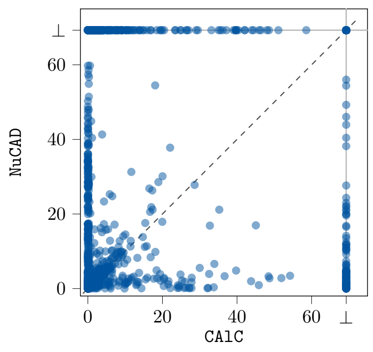


Figure 6: Running time in seconds. $\perp$ indicates a timeout.

Table 1
Statistics of the solvers on instances solved by both solvers. Each row is the sum of the given statistic of the given instances.

	CA1C	NuCAD
# atoms	93 118	142 668
# cells	123 696	166 819
# symbolic intervals	173 378	727 891
# sections	100 492	3 003 788
real root time (s)	70 (17.59%)	4652 (61%)
non-alg. time (s)	3141 (78.39%)	2738 (35.64%)

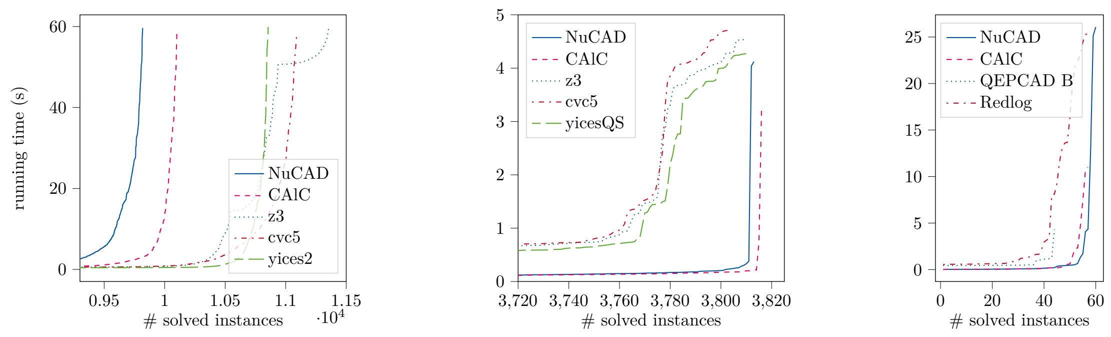
using the MCSAT implementation of SMT-RAT for quantifier-free formulas, as they accept indexed root expressions as input.

5.1. Comparison of NuCAD and CAlC

As the NRA and QE benchmark sets do mostly contain small formulas, we compare NuCAD and CA1C on the QF_NRA benchmark set by treating all variables as parameters, e.g. we generate a formula that describes all solutions by quantifier elimination. Note that the input formula already does this implicitly; however, the results by quantifier elimination describe them by cylindrical cells, which makes e.g. the generation of a solution point straight-forward. Of the 12154 instances, CA1C solved 9559, while NuCAD only solved 9253; CA1C solved 422 not solved by NuCAD, while NuCAD solved 116 not solved by CA1C. Figure 6 shows that CA1C is much faster on the majority of the instances, but there are instances where NuCAD is more efficient. Thus, the algorithms seem complementary to some degree.

Looking at Table 1, CA1C produces smaller solution formulas. Comparing the number of cells (before the simplification) and symbolic intervals that describe the cells, we see that NuCAD requires more cells and proportionally more computation steps (reflected by the number of symbolic intervals). The reason for this clear difference is not obvious: as the CAlC algorithm computes coverings of cylinders (which may lead to repeated expensive computations), it may visit the same cells multiple times, while NuCAD visits each cell only once (sometimes leading to coarser decompositions, e.g. compare Figure 1b and Figure 3).

Further, the CA1C avoids the costly exploration of sections more successfully than NuCAD: Looking again at Table 1, NuCAD explores 29 times more sections than CA1C. As consequence, NuCAD spends significantly more time on real root isolation. We hoped that the optimization in Section A.1 might avoid large parts of the sections. However, on instances solved by both solvers, during the run of NuCAD, only 5% of all non-point intervals have at least one closed bound. For CA1C, which implements a similar technique, this number is 90%.



(a) SMT-LIB QF_NRA.　(b) SMT-LIB NRA.　(c) Wilson QE.

Figure 7: Performance profiles of SMT solvers and QE tools.

Still, NuCAD spends fewer time on non-algebraic computations. In fact, we observe that NuCAD is able to solve bigger formulas (the mean number of nodes in the directed acyclic graph representing the formula is 833 (1565) over the instances solved exclusively by NuCAD (CAlC)), while CAlC is able to solve formulas with higher degree (the mean maximal degree is 3.8 (9.85) over the instances solved exclusively by NuCAD (CAlC)); see also Table 2 in the appendix.

5.2. Comparison with SMT and Quantifier Elimination Tools

We compare NuCAD and CAlC against the state-of-the-art SMT solvers z3 4.13.4, cvc5 1.2.0, and yices2 2.6.5 / yicesQS, as well as the open source quantifier elimination tools QEPCAD B (through Tarski 1.28) and Redlog 6658. The SMT results are depicted in Figure 7a and Figure 7b, the QE results in Figure 7c. We omit the comparison against the TIOpen-NuCAD implementation from [7, 8] as it is incomplete and a fair comparison is not possible on the given benchmarks.

While NuCAD and CAlC are not competitive on QF_NRA, both outperform the other solvers on NRA. On the QE benchmarks, NuCAD even solves the most instances. However, the NRA and QE benchmarks are quite small and not diverse, we thus cannot draw any robust conclusion. Unfortunately, there are no more accessible benchmark collections for these problems available currently.

6. Conclusion

We reported on the first complete implementation of NuCAD. We simplified the algorithm for quantifier alternation by considering the quantifier structure during the computations, and applied further optimizations to reduce the computational effort.

We evaluated the NuCAD algorithm against the related CAlC algorithm. We found that CAlC is more efficient on problems without quantifier elimination, as it produces fewer cells and avoids heavy computations with non-rational numbers more effectively. On the two benchmark sets with quantifier alternations, NuCAD was competitive or even more efficient; however, these sets are too small and not sufficiently diverse for drawing any reliable conclusion. Combinations of the two algorithms could be explored in future to benefit from their individual strengths. In particular, NuCAD is better suited for parallelization: it computes a decomposition, and thus exactly describes the parts of the space which are yet unexplored. We thus can delegate their exploration to threads in a clean way.

Declaration on Generative AI

No generative AI was used.

References

[1] Erika Ábrahám et al. “Deciding the Consistency of Non-Linear Real Arithmetic Constraints with a Conflict Driven Search Using Cylindrical Algebraic Coverings”. In: *Journal of Logical and Algebraic Methods in Programming* 119 (Feb. 2021).

[2] Nikolaj Bjørner and Mikolas Janota. “Playing with Quantified Satisfaction”. In: *Logic for Programming, Artificial Intelligence and Reasoning (LPAR-20)*. Dec. 2015.

[3] Maria Paola Bonacina, Stéphane Graham-Lengrand, and Christophe Vauthier. “QSMA: A New Algorithm for Quantified Satisfiability Modulo Theory and Assignment”. In: *Automated Deduction (CADE-29)*. 2023.

[4] Christopher W. Brown. “Solution Formula Construction for Truth Invariant CAD’s”. PhD thesis. University of Delaware, 1999.

[5] Christopher W. Brown. “QEPCAD B: A Program for Computing with Semi-Algebraic Sets Using CADs”. In: *ACM SIGSAM Bulletin* 37.4 (Dec. 2003).

[6] Christopher W. Brown. “Constructing a Single Open Cell in a Cylindrical Algebraic Decomposition”. In: *Symbolic and Algebraic Computation (ISSAC 2013)*. 2013.

[7] Christopher W. Brown. “Open Non-uniform Cylindrical Algebraic Decompositions”. In: *Symbolic and Algebraic Computation (ISSAC 2015)*. June 2015.

[8] Christopher W. Brown. “Projection and Quantifier Elimination Using Non-uniform Cylindrical Algebraic Decomposition”. In: *Symbolic and Algebraic Computation (ISSAC 2017)*. July 2017.

[9] Christopher W. Brown and Marek Košta. “Constructing a Single Cell in Cylindrical Algebraic Decomposition”. In: *Journal of Symbolic Computation* 70 (Sept. 2015).

[10] Changbo Chen et al. “Computing Cylindrical Algebraic Decomposition via Triangular Decomposition”. In: *Symbolic and Algebraic Computation (ISSAC 2009)*. July 2009.

[11] George E. Collins. “Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition”. In: *Automata Theory and Formal Languages*. 1975.

[12] Florian Corzilius et al. “SMT-RAT: An Open Source C++ Toolbox for Strategic and Parallel SMT Solving”. In: *Theory and Applications of Satisfiability Testing (SAT 2015)*. 2015.

[13] Leonardo de Moura and Nikolaj Bjørner. “Efficient E-Matching for SMT Solvers”. In: *Automated Deduction (CADE-21)*. 2007.

[14] Leonardo de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*. 2008.

[15] Bruno Dutertre. “Yices 2.2”. In: *Computer Aided Verification (CAV 2014)*. 2014.

[16] Hidenao Iwane et al. “An Effective Implementation of a Symbolic-Numeric Cylindrical Algebraic Decomposition for Quantifier Elimination”. In: *Symbolic Numeric Computation (SNC 2009)*. Aug. 2009.

[17] Dejan Jovanović and Leonardo de Moura. “Solving Non-linear Arithmetic”. In: *Automated Reasoning (IJCAR 2012)*. 2012.

[18] Gereon Kremer and Jasper Nalbach. “Cylindrical Algebraic Coverings for Quantifiers”. In: *Satisfiability Checking and Symbolic Computation (SC-square 2022)*. 2023.

[19] Gereon Kremer et al. “On the Implementation of Cylindrical Algebraic Coverings for Satisfiability Modulo Theories Solving”. In: *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2021)*. Dec. 2021.

[20] Jasper Nalbach and Erika Ábrahám. “Merging Adjacent Cells During Single Cell Construction”. In: *Computer Algebra in Scientific Computing (CASC 2024)*. 2024.

[21] Jasper Nalbach and Gereon Kremer. *Extensions of the Cylindrical Algebraic Covering Method for Quantifiers.* 2024.

[22] Jasper Nalbach et al. “Levelwise Construction of a Single Cylindrical Algebraic Cell”. In: *Journal of Symbolic Computation* 123 (July 2024).

[23] Aina Niemetz et al. “Syntax-Guided Quantifier Instantiation”. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2021).* 2021.

[24] Lynn Pickering et al. “Explainable AI Insights for Symbolic Computation: A Case Study on Selecting the Variable Ordering for Cylindrical Algebraic Decomposition”. In: *Journal of Symbolic Computation* 123 (July 2024).

[25] Mathias Preiner et al. *SMT-LIB Release 2024 (Non-Incremental Benchmarks).* 2024.

[26] Andreas Seidl and Thomas Sturm. “A Generic Projection Operator for Partial Cylindrical Algebraic Decomposition”. In: *Symbolic and Algebraic Computation (ISSAC 2003).* Aug. 2003.

[27] Adam Strzeboński. “Solving Systems of Strict Polynomial Inequalities”. In: *Journal of Symbolic Computation* 29.3 (Mar. 2000).

[28] David Wilson. *Real Geometry and Connectedness via Triangular Description: CAD Example Bank.* 2013.

A. Appendix

A.1. Improved Splitting

The loop in Lines 10-18 of Algorithm 5 splits the input cell $(\mathtt{I}_{i+1}, \dots, \mathtt{I}_{i'})$ into the generated cell $(\mathtt{I}'_{i+1}, \dots, \mathtt{I}'_{i'})$ and the neighbouring sections and sectors: if the generated cell is a sector, then we split the space below and above into a section and sector respectively, and explore them separately. However, in many cases, handling them separately causes unnecessary effort, as their truth value may be the same — see e.g. the first and second child of $\mathtt{T}$ in Figure 4. In particular, exploring the sections is computationally heavy, as the points sampled from a section are likely non-rational.

By using half-closed symbolic intervals, we can avoid such unnecessary splitting (or defer splitting in the corresponding recursive call for cases where the truth value indeed changes). For doing so, we adapt Algorithms 5 and 7 as follows:

- We allow closed bounds (e.g. $(l, u]$, $[l, u]$) for the symbolic intervals of the input cell and the generated cell.
- When adding the defining polynomials of the input cell to the polynomial set in Algorithm 5 of Algorithm 5, we flag the ones that stem from closed bounds. For them, we do not require sign-invariance in the required cell, but only *semi-sign-invariance* (it holds $p \leq 0$ or $p \geq 0$ on the required cell), which is introduced in [20].
- During the computation of the cell, we use the techniques described in [20, Section 5] which is able to derive half-closed cell bounds if possible, based on the mentioned flags and other properties.
- We replace the body of the **for**-loop in Algorithm 5 of Algorithm 5 by Algorithm 8.
- Some cells may turn out empty when choosing a sample point; these cells need to be omitted.

Algorithm 8: Modification of `nucad_full()`.

```
1 if $\mathtt{I}'_j.l \neq \mathtt{I}_j.l$ then
2     let $? = ($ if $\mathtt{I}_j.l$ is strict and $? = [$ otherwise
3     if $\mathtt{I}'_j.l$ is weak then
4         $Q := Q \cup \{(\mathtt{I}'_{i+1}, \dots, \mathtt{I}'_{j-1}, ?\mathtt{I}_j.l, \mathtt{I}'_j.l), \mathtt{I}_{j+1}, \dots, \mathtt{I}_{i'})\}$
5     else
6         $Q := Q \cup \{(\mathtt{I}'_{i+1}, \dots, \mathtt{I}'_{j-1}, ?\mathtt{I}_j.l, \mathtt{I}'_j.l], \mathtt{I}_{j+1}, \dots, \mathtt{I}_{i'})\}$
7 if $\mathtt{I}'_j.u \neq \mathtt{I}_j.u$ then
      // analogous
```

A.2. Encoding a NuCAD

For constructing a solution formula for quantifier elimination problems, we need to encode the NuCAD that describes the solution space. For doing so, we apply a post-processing to the NuCAD data structure — similarly as done for the CAlC algorithm in [21]. The idea is — given some node in the NuCAD data structure — (1) to merge all neighbouring children which carry the same label (*true* or *false*) and (2) to replace the subtree of a node where all of its leaves carry the same label by that label.

Example A.1. *Consider Figure 3. The cell* $[\mathrm{root}_{x_1}^{p_5,1}, \mathrm{root}_{x_1}^{p_5,1}] \times (-\infty, \infty)$ *is split into multiple cells which carry the same label, thus they can be merged during post-processing. Similarly, the cells after* $(-\infty, \mathrm{root}_{x_1}^{p_8,1}) \times [\mathrm{root}_{x_2}^{p_4,1}, \mathrm{root}_{x_2}^{p_4,1}]$ *and* $(-\infty, \mathrm{root}_{x_1}^{p_8,1}) \times [\mathrm{root}_{x_2}^{p_1,1}, \mathrm{root}_{x_2}^{p_1,1}]$ *can be merged respectively.* ⌟

For obtaining the solution formula, we encode the parts of the tree that describe the cells labelled with *true*. Without the preprocessing step, we would only need to encode the *outermost* bounds on the variables (i.e. the bounds in the nodes closest to the leaves). However, after the preprocessing, some of these bounds are omitted; we thus need to include additional (non-outermost) bounds on variables. Further, in some cases, we would yield smaller solution formulas by taking the negation of the encoding of the cells labelled with *false*. The work in [21] describes a technique that makes use of this observation and applies it also to subtrees of the NuCAD data structure.

A.3. Evaluation

Table 2
Features of solved instances, split by instances solved by both, none, or one solver exclusively. The features are the number of nodes (# nodes) of the directed acyclic graph that encodes the input formula, and the maximal degree (max degree) of a single variable in a polynomial of the input formula. For each feature, we give the mean, median and maximal value over the corresponding instances.

	instances	# nodes			max degree		
	# solved	mean	median	max	mean	median	max
both	9028	430.08	65.00	6417	5.13	3	44
only CAlC	232	833.12	306.50	4768	9.85	3	94
only NuCAD	145	1565.12	2094.00	3703	3.80	2	16
none	2749	13591.37	1427.00	664211	2.64	2	22