J. Nalbach, G. Kremer
Math Comput Sci(2026)20:7
https://doi.org/10.1007/s11786-025-00623-3

Mathematics in Computer Science

RESEARCH

Extensions of the Cylindrical Algebraic Covering Method for Quantifiers

Jasper Nalbach[1*] and Gereon Kremer[2]

[*]Correspondence:
nalbach@cs.rwth-aachen.de
[1]RWTH Aachen University, Aachen, Germany
[2]Certora Ltd, Tel Aviv, Israel
Communicated by Christopher W. Brown

Abstract

The *cylindrical algebraic covering* method was originally proposed to decide the satisfiability of a set of *non-linear real arithmetic* constraints. We reformulate and extend the cylindrical algebraic covering method to allow for checking the truth of arbitrary non-linear arithmetic formulas, adding support for both quantifiers and Boolean structure. Furthermore, we also propose a variant to perform *quantifier elimination* on such formulas. After introducing the algorithm, we elaborate on various extensions, optimizations and heuristics. Finally, we present an experimental evaluation of our implementation and provide a comparison with state-of-the-art SMT solvers and quantifier elimination tools.

Keywords: Non-linear arithmetic, Cylindrical algebraic covering, Quantifier elimination

Mathematics Subject Classification: Primary 14Q99, Secondary 68W99

1 Introduction

Non-linear real arithmetic (NRA) (or *real algebra*) is the first-order theory whose atoms are polynomial constraints over real variables. We consider three fundamental questions with regard to this theory: (1) *Satisfiability* of quantifier-free formulas; that is, deciding whether an assignment to the formula's variables exists such that the formula evaluates to `True`. (2) *Truth of sentences*; that is, deciding whether formulas where all variables are quantified are equivalent to `True` or `False`. Satisfiability is a special case of this question, as we can existentially quantify all free variables to obtain a sentence. (3) *Quantifier elimination* in formulas containing both free variables (*parameters*) and quantified variables; that is, computing an equivalent quantifier-free formula over the parameters. Deciding the truth of sentences is a special case of this question, as we can eliminate all quantifiers.

The field of *satisfiability-modulo-theories (SMT)* solving deals with the first two problems; while checking the satisfiability of quantifier-free non-linear real arithmetic formula has fairly good support, checking the truth of sentences still lacks accessible and efficient tools. The tools for *quantifier elimination* work a bit differently from SMT tools and might benefit from integrating ideas from SMT solving.

Tarski [48] established the existence of quantifier elimination methods for non-linear real arithmetic, although his method was practically unusable due its non-elementary

© The Author(s) 2026. This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

complexity bounds. Today, the *cylindrical algebraic decomposition (CAD)* [20] method is the only complete procedure for answering all these questions that is used in practice, despite its doubly exponential worst-case complexity that severely limits the scalability of the method. For the satisfiability problem of conjunctions of constraints, motivated by the application in SMT solving, the *cylindrical algebraic covering (CAlC)* method [3] has been developed based on cylindrical algebraic decomposition. Although it retains the doubly exponential complexity, its performance is significantly better in practice [3,33] while its implementation requires only a simple bookkeeping data structure. Furthermore, it more closely resembles human reasoning and is more amenable to proof production [2,4].

Contribution This paper extends [34] in which we propose a novel reformulation and extension of the cylindrical algebraic covering method that goes beyond the satisfiability problem of conjunctions to allow solving arbitrary quantified formulas as well as quantifier elimination queries. This work elaborates the details of the algorithm, proposes some optimizations and extensions, and provides an evaluation of its implementation. We first consider checking truth where all variables are explicitly quantified, either existentially or universally, in Sect. 4, and then expand to the *quantifier elimination* problem in Sect. 5. The presented method does not rely on a SAT solver to solve a Boolean abstraction; we will elaborate in Sect. 6 how the Boolean structure is incorporated. Afterwards, we present a divide-and-conquer adaption in Sect. 7 and a fine-grained proof system for CAlC in Sect. 8. Finally, we elaborate on heuristics for directing the search in Section 9.1 and evaluate them in Sect. 9.2, comparing with state-of-the-art tools. We conclude in Section 10.

2 Related Work

2.1 Quantifier-Free Formulas

The *NLSAT* [30] algorithm, later generalized to the *model-constructing satisfiability calculus (MCSAT)* [23] and implemented in `z3` and `yices2`, as well as in our solver `SMT-RAT`, can be seen as an extension of CDCL(T) [26] for first-order theories; it is thus a combination of a search for a satisfying solution (exploration) and generalization of unsatisfiable solutions (deduction). In addition to the Boolean reasoning, we make decisions and propagations on theory variables as well, i.e., we assign values to theory variables and then evaluate and propagate literals based on these assignments. The assignment of theory variables is required to not violate any theory constraint that is assumed to be true on the Boolean level. If such an assignment is not possible for a theory variable x_{n+1}, then the unsatisfying intervals of the theory constraints (partially evaluated using the values for $x_1, \ldots, x_n$) cover the whole real line $\mathbb{R}$. This conflict is generalized using CAD-based techniques to a formula that excludes a connected set S in $\mathbb{R}^n$ such that the cylinder $S \times \mathbb{R}$ is covered by unsatisfying sets of the same constraints.

The *cylindrical algebraic covering (CAlC)* method [3] is implemented both in `cvc5` and in `SMT-RAT`. It has has a number of similarities to the NLSAT algorithm, but is a theory solver for the classical CDCL(T) architecture and thus only considers conjunctions of constraints instead of arbitrary formulas. The bookkeeping required is simpler than for MCSAT: if a conflict occurs, it is not generalized to a formula in $x_1, \ldots, x_n$, but uses an implicit description of the unsatisfying values, which consists of a set of CAD projection polynomials and the sample $s \in \mathbb{R}^n$ that caused the conflict; this implicit description

is used to deduce unsatisfying intervals, similarly to how they are gathered from the input constraints. While MCSAT allows for both breadth-first and depth-first search (or a combination of both), CAlC is restricted to a depth-first search due to its simpler bookkeeping.

2.2 Quantified Formulas

To decide sentences including quantifiers, the `z3` and `cvc5` solvers implement approaches based on *quantifier instantiation* [9,22,27,44] which are *incomplete* for non-linear real arithmetic. We do not go into detail here, as these approaches work differently from our algorithm; for an overview, we refer to the related work section in [44].

A recent *complete* algorithm is the *QSMA* algorithm [10] implemented in `yicesQS`. It maintains for every quantified subformula $Qy.\varphi(x_1, \ldots, x_n, y)$ (where $Q \in \{\exists, \forall\}$) an over- and under-approximation (encoded as formulas in the free variables $x_1, \ldots, x_n, y$ of φ) which are refined iteratively in a breadth-first search manner. This refinement continues until these approximations are fine-grained enough to conclude satisfiability or unsatisfiability. The required approximations can be generated by NLSAT-based solvers.

For quantifier elimination, mentionable tools implementing CAD that are readily available include `QEPCAD B` [12,15,16] and `Redlog` [46]. In particular, the former applies various techniques to reduce the number of projection polynomials in the CAD used to construct the solution formulas, which is often coarser than the result of naive CAD algorithms. Furthermore, the commercial tools `Maple` [29] and `Mathematica` [47] implement the CAD for quantifier elimination. We refer to the respective sources for more details.

3 Preliminaries

Let $s \in \mathbb{R}^i$, $s' \in \mathbb{R}$, and $I \subseteq \mathbb{R}$. We denote the Cartesian product of two sets using the $\times$ symbol. We denote by $s \times s'$ the point $(s_1, \ldots, s_i, s') \in \mathbb{R}^{i+1}$, by $s_{[j]}$, $j < i$ the point $(s_1, \ldots, s_j) \in \mathbb{R}^j$, and by $s \times I$ the Cartesian product $\{s\} \times I \subseteq \mathbb{R}^{i+1}$. Let $R \subseteq \mathbb{R}^i$, then $R{\downarrow_{[j]}}$, $j < i$ denotes the projection of R onto the first j coordinates. A set $R \subseteq \mathbb{R}^i$ is called a *cell* if it is connected. Given a tuple such as $t = (a, b, c)$, we use $t.a$, $t.b$ and $t.c$ to access its entries.

Let $x_1, \ldots, x_n$ be *variables*, then $\mathbb{Q}[x_1, \ldots, x_i]$ denotes the set of all *polynomials* with rational coefficients in $x_1, \ldots, x_i$. We assume an ordering on the variables $x_1 \prec \cdots \prec x_n$. The highest variable (w.r.t. this ordering) occurring in a polynomial p is called the *main variable* and its *level* level(p) is the index of the main variable. The *degree* of p in x_i is denoted by $\deg_{x_i}(p)$. Let $s \in \mathbb{R}^j$, $j \leq i$, then $p(s, x_{j+1}, \ldots, x_i)$ denotes the polynomial after substituting s into p. The polynomial p is called *nullified* over s if $j < i$ and $p(s, x_{j+1}, \ldots, x_i) = 0$. Let $p \in \mathbb{Q}[y]$ be a univariate polynomial, then realRoots(p) denotes the set of *real roots* of p in y.

A polynomial constraint $p \sim 0$ compares a polynomial p with zero using a relation symbol $\sim \in \{=, \leq, \geq, \neq, <, >\}$; notations for polynomials are transferred to constraints where meaningful. We assume every formula φ to be a first-order formula over non-linear real arithmetic with polynomial constraints defined in variables $x_1, \ldots, x_n$. We use $\equiv$ do denote equivalence of formulas (modulo non-linear real arithmetic). A cell $R \subseteq \mathbb{R}^i$ is called *semi-algebraic* if it is the solution set of a quantifier-free non-linear real arithmetic

formula. A polynomial p is called *sign-invariant* on R if for all points in R the evaluation of p has the same sign. A formula φ is called *truth-invariant* on R if for all points in R the evaluation of φ is equivalent to the same truth value.

Let $s \in \mathbb{R}^i$, then we denote the *(partial) evaluation up to level i* of φ over s by $\varphi[s]$: constraints of level i evaluate to `True` or `False` according to standard semantics, otherwise they evaluate to `Undef` (i.e., under this partial evaluation $x_1 \cdot x_2 > 1$ evaluates to `Undef` at $x_1 = 0$). The semantics are extended for formulas inductively according to the three-state semantics of the logical operators; that means, we evaluate formulas also on sample points which do not assign all variables in the formula, e.g. "`Undef` $\wedge$ `False`" evaluates to `False`.

A formula φ is in *prenex normal form* if it consists of a *prefix* of quantifiers and a quantifier-free formula called the *matrix* $\overline{\varphi}$:

$$\varphi := Q_{k+1}x_{k+1} \cdots Q_n x_n.\ \overline{\varphi}(x_1, \ldots, x_n)$$

If $k \neq 0$, φ has free variables (also called *parameters*) that are not explicitly quantified.

3.1 Cylindrical Algebraic Decomposition

We give a short introduction to CAD. For the understanding of some details of this paper, we assume the reader to be familiar with CAD (we refer for instance to [32]). We first formally define what the output of a CAD computation is:

Definition 3.1 *(Cylindrical Algebraic Decomposition)* Let $i \in \mathbb{N}_{>0},\ i \leq n$.

Let $R \subseteq \mathbb{R}^i$ be a cell. R is called *locally cylindrical* if either $i = 1$; or $i > 1$, $R{\downarrow_{[i-1]}}$ is locally cylindrical, and there exist continuous functions $\theta_1, \theta_2 : R{\downarrow_{[i-1]}} \to \mathbb{R}$ such that either $R = \{(r, \theta_1(r)) \mid r \in R{\downarrow_{[i-1]}}\}$ (R is called a *sector* over $R{\downarrow_{[i-1]}}$) or $R = \{(r, r') \mid r \in R{\downarrow_{[i-1]}}, \theta_1(r) < r' < \theta_2(r)\}$ (R is called a *section* over $R{\downarrow_{[i-1]}}$).

Let $D \subset \{R \mid R \subseteq \mathbb{R}^i\}$ be finite. D is called a *decomposition* if $\cup_{R \in D} R = \mathbb{R}^i$ and $R \cap R' = \emptyset$ for all $R, R' \in D,\ R \neq R'$. D is called a *cylindrical decomposition* if D is a decomposition, each $R \in D$ is locally cylindrical; and either $i = 1$, or $i > 1$ and $D' := \{R{\downarrow_{[i-1]}} \mid R \in D\}$ is a cylindrical decomposition. D is called a *cylindrical algebraic decomposition (CAD)* if it is a cylindrical decomposition and each $R \in D$ is semi-algebraic.

A CAD D is computed such that it is *sign-invariant for* $P \subset \mathbb{Q}[x_1, \ldots, x_n]$, i.e. every $p \in P$ is sign-invariant in every $R \in D$. A CAD is useful because of its cylindrical structure: Firstly, local cylindricity of each cell allows to iteratively compute a point that is contained in the cell. Secondly, the cylindrical arrangement of the cells (that is, given a cell $R' \in D'$, we know that there are $R_1, \ldots, R_k \in D$ such that the cylinder $R' \times \mathbb{R}$ over R' is equal to $R_1 \cup \ldots \cup R_k$) allows us to systematically explore the decomposition and reason about its structure.

A sign-invariant CAD is computed recursively: Given a set $P \subset \mathbb{Q}[x_1, \ldots, x_i]$ of polynomials, we compute a projection $P' \subset \mathbb{Q}[x_1, \ldots, x_{i-1}]$ such that, given a sign-invariant CAD D' of P', we can compute a CAD D for P. To guarantee the latter, we define the projection such that the set P is *delineable* on each cell $R' \in D'$, that is, the roots $\{(r', r) \mid r' \in R', r \in \text{realRoots}(p(r', x_i)), p \in P\}$ of P on R' can be described by continuous *root functions* $\theta_1, \ldots, \theta_k : R' \to \mathbb{R}$ such that $\theta_1(r') < \ldots < \theta_k(r')$ for all $r' \in R'$, i.e.

these roots do not intersect. These functions then define the bounds of the cells in D that decompose the cylinder $R' \times \mathbb{R}$.

A *projection operator* maps polynomials of level i to polynomials of lower level. Ingredients are *discriminants* $\text{disc}_{x_i}(p)$ which—together with some *coefficients* $\text{coeff}_{x_i}(p)$—guarantee delineability of a single polynomial $p \in \mathbb{Q}[x_1, \ldots, x_i]$ (i.e. the root functions of p are well-defined on R' and do not intersect), and *resultants* $\text{res}_{x_i}(p, q)$ which guarantee that the root functions of two polynomials $p, q \in \mathbb{Q}[x_1, \ldots, x_i]$ are either equal or do not intersect on R'.

The CAD algorithm has doubly exponential complexity [17] due to the computation of iterated resultants (and discriminants) which results in a doubly exponential growth of degrees [25]. While this growth is inherent to the CAD, different projection operators with vastly different output sizes exist. If a projection operator computes more polynomials than necessary, the subsequent projection steps and computations oftentimes amplify this overhead drastically and thereby impact the efficiency in practice.

The original operator by Collins [20] is complete but expensive. McCallum's projection operator [38] is more efficient but incomplete (and was later improved by Brown [13]), i.e., it fails if any polynomial in the projection is nullified over some point. It also does not compute a sign-invariant CAD, but an *order-invariant* CAD, which is a property of polynomials which is stronger than sign-invariance (as the formal details are not required for this paper, we refer to [38] for them). Lately, Lazard's projection operator [36], which is complete and similarly efficient as McCallum, has been proven to be correct [39]. Although this work is applicable to all mentioned projection operators, the projection operator defined in this paper is based on McCallum's projection due to its efficiency and simplicity; the presented algorithm detects nullifications and returns an unknown result.

3.2 Cylindrical Algebraic Covering

A CAD already allows to answer our questions about a non-linear real arithmetic formula φ: To do so, we compute a sign-invariant CAD D for all polynomials occurring in φ; D will be truth-invariant for φ. However, the CAD is finer than we actually need. We thus define the notion of *cylindrical algebraic covering (CAlC)*, which is defined analogously to Definition 3.1, but does not require that its cells are disjoint. We now compute a covering C that is truth-invariant for φ, but not necessarily sign-invariant for its defining polynomials!

When computing a CAD, we fully decompose the space top-down, i.e., compute a full projection, and then construct the resulting cells. The CAlC is computed bottom-up: We compute sample points and generalize the reasons for the truth-value of φ at these points.

We briefly present the idea behind the CAlC method for checking the *existential fragment* of non-linear real arithmetic (i.e. the input is in prenex normal form and all variables are existentially quantified) and refer to [3] for more details and to the preliminaries of [6] for another intuitive introduction.

The fundamental idea is to recursively construct a (partial) sample point and collect intervals that represent unsatisfiable cells above this sample point. When a sample point can not be extended because these intervals form a covering of the real line in the next dimension, the covering is projected into the previous dimension to refute the current sample point. We then backtrack and choose a different value for the variable on the

highest level. Eventually, either a full sample point is constructed, and we return SAT; or an unsatisfiable covering is constructed in the first dimension, and we return UNSAT. In contrast to cells from cylindrical algebraic decomposition, intervals do not form a decomposition as they may overlap.

The algorithm starts by constructing unsatisfiable intervals for x_1 based on univariate constraints and then tries to select a value s_1 for the variable x_1 outside these intervals. If such a value exists, the method is called recursively with the partial sample point (s_1). After substituting $x_1 = s_1$, the constraints with main variable x_2 become univariate and thus suitable for identifying unsatisfiable intervals for x_2. This process is continued recursively until either all constraints are satisfied (and we return SAT) or for some x_i no suitable value exists. In the latter case, the set of unsatisfiable intervals covers the whole real line and forms a covering. This covering is generalized by projecting it to dimension $i-1$. The idea is to use projection tools borrowed from cylindrical algebraic decomposition with some improvements: as we only need to characterize this covering and not a decomposition, only a subset of the full projection is needed. Using the current sample point, an interval for the variable x_{i-1} with respect to the projection result can be computed which is added to the set of unsatisfiable intervals for x_{i-1}, possibly taking part in an unsatisfiable covering for x_{i-1}. We now try another value for x_{i-1}, respecting the current set if unsatisfiable intervals. Unless we find a full satisfying sample point we eventually obtain an unsatisfiable covering for the first variable x_1 and return UNSAT.

Implicit Cells. We generalize intervals (over a partial sample point) by attaching algebraic information in the form of sets of polynomials whose order-invariance characterizes satisfiability-invariant cells of a multivariate formula.

Definition 3.2 *(Implicit Cell)* Let $i \in \mathbb{N}_{>0}$, $P \subseteq \mathbb{Q}[x_1, \ldots, x_i]$ be a set of polynomials, $s \in \mathbb{R}^i$, and $I \subseteq \mathbb{R}$ be an interval. Let $R \subseteq \mathbb{R}^i$ be the maximal connected subset containing s where all polynomials in P are order-invariant. An *implicit cell of level i* is a tuple (P, s, I) such that $I = \{r \mid (s_1, \ldots, s_{i-1}, r) \in R\}$, i.e., its i-th coordinate is bounded from below by is the greatest root of P in x_i below (or equal to) s_i and bounded from above by the smallest root above (or equal to) s_i. R is called the *cell defined by P and s*.

Example Consider the polynomials $P = \{x_2 + 1, x_1^2 + x_2^2 - 2, x_1 - 1\}$ and the sample point $s = (0, 0)$. Then the corresponding implicit cell is $(P, s, (-1, \sqrt{2}))$, as $x_2 + 1$ has a zero at $(0, -1)$, $x_1^2 + x_2^2 - 2$ at $(0, \sqrt{2})$, and no polynomial has a zero over $(s_1) = (0)$ between these two points.

Implicants. For reasoning about the Boolean structure of a formula, we introduce the notion of implicants. An *implicant* ψ of a quantifier-free formula φ is usually understood to be a "simpler" quantifier-free formula that implies φ ($\psi \Rightarrow \varphi$), and the set of constraints of ψ is a subset of the ones in φ. We adapt this concept as follows.

Definition 3.3 *(Implicant)* Let $s \in \mathbb{R}^i$ be a (partial) sample point and φ be a quantifier-free formula in n variables. If $\varphi[s] \equiv \texttt{True}$, then ψ is an *implicant of φ with respect to s* if

$$\psi[s] = \texttt{True} \wedge (\psi \Rightarrow \varphi)$$

and the constraints of ψ are of level at most i and contained in φ. If $\varphi[s] \equiv \texttt{False}$, then ψ is an *implicant of φ with respect to s* if

$$\psi[s] = \texttt{True} \wedge (\psi \Rightarrow \neg\varphi)$$

and the constraints of ψ are of level at most i and contained in φ. We call ψ a *prime implicant* of φ if the set of constraints in ψ is minimal among all implicants of φ.

Note that in the above definition, we allow $\varphi[s]$ to be a tautology or contradiction, while we require that the implicant $\psi[s]$ evaluates to a truth value after plugging in s.

Example Let $\varphi = x^2 > 0 \wedge (x < 2 \vee x > 4)$. Note that $\varphi(1) = \texttt{True}$, $\varphi(3) = \texttt{False}$ and $\varphi(0) = \texttt{False}$. $x^2 > 0 \wedge x < 2$ is a prime implicant of φ w.r.t. 1. $\neg(x < 2 \vee x > 4)$ is a prime implicant of φ w.r.t. 3. Both $\neg(x^2 > 0)$ and $\neg(x^2 > 0 \wedge x > 4)$ are implicants of φ w.r.t. 0, but only the first is a prime implicant.

Let $\varphi = (x < 0 \vee y \leq 4) \wedge (x > 2 \vee y > 4)$. Note that $\varphi(1, y) \equiv \texttt{False}$. $\neg(x < 0) \wedge \neg(x > 2)$ is a prime implicant of φ w.r.t. 1.

4 Quantified Problems

We first describe how the cylindrical algebraic covering method can be adapted for problems where all variables are quantified (again, assuming the input formula is in prenex normal form). Our presentation follows the structure of [3], but is different in some details.

Algorithm 1: `user_call()`

Data : Global prefix $Q_1 x_1 \cdots Q_n x_n$ and matrix $\overline{\varphi}$.
Output : Either `SAT` or `UNSAT`

```
1 (f, O) := recurse(())                                   // Algorithm 2
2 return f
```

Algorithm 2: `recurse(`s`)`

Data : Global prefix $Q_1 x_1 \cdots Q_n x_n$ and matrix $\overline{\varphi}$.
Input : Sample point $s = (s_1, \ldots, s_{i-1}) \in \mathbb{R}^{i-1}$.
Output : (`SAT`, C) or (`UNSAT`, C) where $s \times C.I$ can or can not be extended to a model for any $s_i \in C.I$. In both cases, C describes how s can be generalized.

```
1 if Q_i = ∃ then return exists(s)                        // Algorithm 3
2 else return forall(s)                                   // Algorithm 4
```

4.1 Existential Quantification

First assume that all variables are existentially quantified, thus our algorithm resembles the original from [3]: Algorithm 3 (which corresponds to `get_unsat_cover` from [3]) is recursively called to choose a suitable value s_i for the next variable x_i such that the resulting sample point does not conflict with the formula $\overline{\varphi}$. The algorithm maintains a list of cells $\mathbb{I}_{\text{unsat}}$ in $\mathbb{R}^i$ that are known to violate the formula. The call to `sample_outside` in Line 3 chooses a value outside of $\mathbb{I}_{\text{unsat}}$.

If $s \times s_i$ immediately evaluates $\overline{\varphi}$ to `False`, we use Algorithm 6 that generalizes this unsatisfying sample to an unsatisfying cell in $\mathbb{R}^i$ (see below for details); in [3], such cells

Algorithm 3: `exists(`s`)`

Data : Global prefix $Q_1 x_1 \cdots Q_n x_n$ and matrix $\overline{\varphi}$.
Input : Sample point $s = (s_1, \ldots, s_{i-1}) \in \mathbb{R}^{i-1}$.
Output : see Algorithm 2

$\mathbb{I}_{\text{unsat}} := \emptyset$
while $\bigcup_{C \in \mathbb{I}_{\text{unsat}}} C.I \neq \mathbb{R}$ **do**
  $s_i :=$ `sample_outside(`$\mathbb{I}_{\text{unsat}}$`)`
  if $\overline{\varphi}[s \times s_i] =$ `False` **then**
    $(f, O) :=$ (`UNSAT`, `get_enclosing_cell(`$s \times s_i$`)`) // Algorithm 6
  else if $\overline{\varphi}[s \times s_i] =$ `True` **then**
    $(f, O) :=$ (`SAT`, `get_enclosing_cell(`$s \times s_i$`)`) // Algorithm 6
  else it holds $i < n$
    $(f, O) :=$ `recurse(`$s \times s_i$`)` // Algorithm 2, recursive call
  if $f =$ `SAT` **then**
    $C :=$ `characterize_cell(`s, O`)` // Algorithm 7
    return (`SAT`, C)
  else if $f =$ `UNSAT` **then**
    $\mathbb{I}_{\text{unsat}} := \mathbb{I}_{\text{unsat}} \cup \{O\}$
$C :=$ `characterize_covering(`$s, \mathbb{I}_{\text{unsat}}$`)` // Algorithm 8
return (`UNSAT`, C)

would be computed by `get_unsat_intervals`. If $s \times s_i$ immediately evaluates $\overline{\varphi}$ to `True`, the method generalizes the satisfying sample to a satisfying cell in $\mathbb{R}^i$; in [3] we would return the satisfying sample only. If the formula does not evaluate to a truth value, we pick a value for the next variable x_{i+1} by a recursive call. If the recursive call returns a satisfying cell $R \subseteq \mathbb{R}^i$, we compute its (CAD-style) projection $R' \subseteq \mathbb{R}^{i-1}$ suitable for the caller (which searches for a value for x_{i-1}) using Algorithm 7. If the recursive call returns an unsatisfying cell, we add it to the list $\mathbb{I}_{\text{unsat}}$. The differences to the original algorithm are due to the support for quantifier alternations, for which we need to generalize both satisfying and unsatisfying sample points.

If the list of unsatisfying cells covers the whole real line $s \times \mathbb{R}$ above the given sample point $s \in \mathbb{R}^{i-1}$, we compute the projection $R' \subseteq \mathbb{R}^{i-1}$ (being an unsatisfiable cell on the lower levels) of a cylinder $R' \times \mathbb{R}$ covered by unsatisfiable cells in $\mathbb{R}^i$ represented by $\mathbb{I}_{\text{unsat}}$ using Algorithm 8.

We note that if $i = 1$, Algorithms 7 and 8 would need to return a cell on the "zero-th level". To simplify the presentation, we assume that a special placeholder value is returned instead of an actual interval.

We emphasize that compared to `get_unsat_cover` from [3], there are no significant algorithmical differences; we mere changed some details such that the algorithm can be extended more easily for quantifier alternation.

4.2 Universal Quantification

Now assume the formula contains quantifier alternations. Algorithm 1 is the interface to the recursive Algorithm 2, calling it with an empty sample point and extracting the main return value. Algorithm 2 checks the current quantifier and calls out to Algorithms 3 or 4 accordingly.

Algorithm 4 is mostly identical to Algorithm 3: While Algorithm 3 collects unsatisfiable cells and returns early when it finds a satisfiable cell, Algorithm 4 collects satisfiable cells and returns early when it finds an unsatisfiable cell. Note that we project cells and coverings

Algorithm 4: `forall(`s`)`

Data : Global prefix $Q_1 x_1 \cdots Q_n x_n$ and matrix $\overline{\varphi}$.
Input : Sample point $s = (s_1, \ldots, s_{i-1}) \in \mathbb{R}^{i-1}$.
Output : see Algorithm 2

$\mathbb{I}_{\text{sat}} := \emptyset$
while $\bigcup_{C \in \mathbb{I}_{\text{sat}}} C.I \neq \mathbb{R}$ **do**
  $s_i :=$ `sample_outside(`$\mathbb{I}_{\text{sat}}$`)`
  if $\overline{\varphi}[s \times s_i] =$ `False` **then**
    $(f, O) :=$ (`UNSAT`, `get_enclosing_cell(`$s \times s_i$`)`) // Algorithm 6
  else if $\overline{\varphi}[s \times s_i] =$ `True` **then**
    $(f, O) :=$ (`SAT`, `get_enclosing_cell(`$s \times s_i$`)`) // Algorithm 6
  else it holds $i < n$
    $(f, O) :=$ `recurse(`$s \times s_i$`)` // Algorithm 2, recursive call
  if $f =$ `SAT` **then**
    $\mathbb{I}_{\text{sat}} := \mathbb{I}_{\text{sat}} \cup \{O\}$
  else if $f =$ `UNSAT` **then**
    $C :=$ `characterize_cell(`s, O`)` // Algorithm 7
    return (`UNSAT`, C)
$C :=$ `characterize_covering(`$s, \mathbb{I}_{\text{sat}}$`)` // Algorithm 8
return (`SAT`, C)

Algorithm 5: `compute_cell(`$s,\ P$`)`

Data : Global matrix $\overline{\varphi}$.
Input : Sample point $s \in \mathbb{R}^i$ and polynomials $P \subseteq \mathbb{Q}[x_1, \ldots, x_i]$.
Output : A maximal interval $I \subseteq \mathbb{R}$ such that P is sign-invariant in $s \times I$.

if $\exists p \in P.\ (\text{level}(p) = i \wedge p(s_{[i-1]}, x_i) = 0)$ **then fail** // We fail on nullification
$Z := \{-\infty, \infty\} \cup \bigcup_{p \in P, \text{level}(p) = i} \text{realRoots}(p(s_{[i-1]}, x_i))$
if $s_i \in Z$ **then return** $[s_i, s_i]$
$l := \max\{z \in Z \mid z \leq s_i\}$
$u := \min\{z \in Z \mid z \geq s_i\}$
return (l, u)

Algorithm 6: `get_enclosing_cell(`s`)`

Data : Global matrix $\overline{\varphi}$.
Input : Sample point $s \in \mathbb{R}^i$ such that $\overline{\varphi}[s] \equiv$ `False` or $\overline{\varphi}[s] \equiv$ `True`.
Output : A satisfiability-invariant implicit cell C containing s.

$P :=$ `implicant_polynomials(`$\overline{\varphi}, s$`)`
replace P by its irreducible factors
$C :=$ ($P, s,$ `compute_cell(`s, P`)`) // Algorithm 5
return C

of cylinders (i.e. calling Algorithms 7 and 8) for both satisfiable and unsatisfiable coverings in the very same way. That is, the projection operations work with implicit cells; whether the input formula is `True` or `False` on an implicit cell is irrelevant for the projection operations.

4.3 Truth-Invariant Cells

Algorithm 6 computes an implicit cell around the given sample point that is satisfiability-invariant with respect to $\overline{\varphi}$. It first obtains the set of polynomials from an implicant of $\overline{\varphi}$ w.r.t. the current sample s by calling `implicant_polynomials`; the sign-invariance of these polynomials directly implies the truth-invariance of $\overline{\varphi}$. The algorithm then uses Algorithm 5 (which corresponds to `interval_from_characterization` in [3]) to construct the interval above s that is contained in a sign-invariant (or truth-invariant) cell.

Algorithm 7: `characterize_cell(`s`, `C`)`

Input : Sample point $s \in \mathbb{R}^i$ and an implicit cell $C = (\cdot, s \times \cdot, \cdot)$ of level $i+1$.
Output : A satisfiability-invariant implicit cell C' containing s.

1 $P_{i+1} := \{p \in P \mid \text{level}(p) = i+1\}$, $P_\perp := P \setminus P_{i+1}$
2 $P' := P_\perp \cup \{\text{disc}_{x_{i+1}}(p) \mid p \in P_{i+1}\} \cup \bigcup_{p \in P_{i+1}} \text{coeff}_{x_{i+1}}(p)$
3 $P' := P' \cup \{\text{res}_{x_{i+1}}(p,q) \mid p,q \in P_{i+1}, p(s \times C.I.l) = 0, \exists s' \leq C.I.l.\ q(s \times s') = 0\}$
4 $P' := P' \cup \{\text{res}_{x_{i+1}}(p,q) \mid p,q \in P_{i+1}, p(s \times C.I.u) = 0, \exists s' \geq C.I.u.\ q(s \times s') = 0\}$
5 $P' := P' \cup \{\text{res}_{x_{i+1}}(p,q) \mid p,q \in P_{i+1}, p(s \times C.I.l) = 0, q(s \times C.I.u) = 0\}$
6 **replace** P' by its irreducible factors
7 **return** $(P', s,$ `compute_cell(`s, P'`)`$)$ `// Algorithm 5`

Algorithm 8: `characterize_covering(`s`, `$\mathbb{I}$`)`

Input : Sample point $s \in \mathbb{R}^i$ and a set $\mathbb{I}$ of implicit cells of level $i+1$ such that $\bigcup_{C \in \mathbb{I}} C.I = \mathbb{R}$ and for all $C \in \mathbb{I}$ it holds $C = (\cdot, s \times \cdot, \cdot)$.
Output : A satisfiability-invariant implicit cell C' of level i containing s.

1 $(C_1, \ldots, C_k) :=$ `compute_cover(`$\mathbb{I}$`)` `// [3, Section 4.4.1]`
2 $P' := \bigcup_{j \in \{1,\ldots,k\}}$ `characterize_cell(`s, C_j`).`P `// Algorithm 7`
3 **for** $j \in \{1, \ldots, k-1\}$ **do**
4 $\quad P' := P' \cup \{\text{res}_{x_{i+1}}(p,q) \mid p \in C_j.P, p(s \times C_j.I.u) = 0, q \in C_{j+1}.P, q(s \times C_{j+1}.I.l) = 0\}$
5 **replace** P' by its irreducible factors
6 **return** $(P', s,$ `compute_cell(`s, P'`)`$)$ `// Algorithm 5`

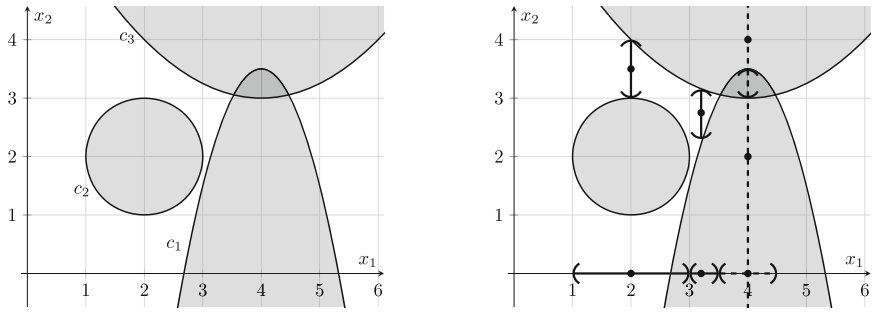
The helper function `implicant_polynomials` is expected to return the polynomials of a (possibly prime) implicant of $\overline{\varphi}$ with respect to s. This might include polynomials not only with main variable x_i, but also lower-level polynomial, effectively bounding also lower-level coordinates of the sign-invariant cell.

If $\overline{\varphi}[s] =$ `False` and $\overline{\varphi}$ is a simple conjunction, it is easy to obtain a *prime implicant* as the negation of a single conflicting constraint in $\overline{\varphi}$; calling it in a loop as done in Algorithm 3 thus emulates `get_unsat_intervals` from [3]. If $\overline{\varphi}[s] =$ `True` and $\overline{\varphi}$ is a simple conjunction and non-redundant (i.e. no sub-formula of $\overline{\varphi}$ implies $\overline{\varphi}$), then $\overline{\varphi}$ itself is the only prime implicant.

Algorithms 7 and 8 implement a reduced CAD projection based on McCallum's projection operator (see Sect. 4.4 for details); these algorithms define the same projection as [3, Algorithm 4], but split the projection into the characterization of individual cells and the characterization of a covering of a cylinder. Algorithm 7 computes a CAD projection $R \subseteq \mathbb{R}^i$ of a single cell in $\mathbb{R}^{i+1}$ and uses Algorithm 5 to construct an interval for x_i. Algorithm 8 first calls the auxiliary method `compute_cover` which takes a set of cells $\mathbb{I}$ as input, and returns a sequence of a subset of these cells: Firstly, it iteratively eliminates "redundant" cells (a cell C is redundant in $\mathbb{I}$ if $C.I \subseteq C'.I$ for some other interval $C' \in \mathbb{I}$); secondly, it sorts the cells by their interval's lower bound. This way, it is guaranteed that neighboring cells in the resulting sequence overlap (or their union is connected); this is important for the correctness of the projection in Line 4. For more details, we refer to [3, Section 4.4.1].

4.4 Details of the Projection Operator

We changed how we normalize the polynomial sets after projection: While [3] assumes "standard CAD simplifications" of the polynomial sets, we explicitly use the set of their irreducible factors in Algorithms 6, 7, and 8 to satisfy the requirements of the projection



(A) Graphs of the constraints. The gray areas depict the conflicting cells of the constraints.

(B) The satisfiable intervals are indicated with a solid line, the unsatisfiable intervals with a dashed line.

Fig. 1 Illustration of the example

operator. Merely using an irreducible square-free basis, the common standard formulation for CAD projection, is not quite sufficient for cylindrical algebraic covering: we eventually compute resultants of polynomials that come from different local projection sets, i.e. from different bases. If carefully executed, these sets can be made "pairwise square-free", as mentioned in [35, Section 2.1]. Fully factoring all polynomials is more robust and probably even more efficient in practice, if the implementation at hand has this capability.

Line 2 of Algorithm 7 adds all coefficients of all polynomials to the projection. [3, Algorithm 6] proposes an optimization, which adds fewer coefficients and could also be applied here.

We further note that the presented projection is based on McCallum's projection operator [38]. An adaption of the projection operator to Lazard's projection [36,39] is possible, as discussed in [3, Section 4.4.6]; this requires an adaption of the added coefficients and root isolation, and changes the correctness arguments (e.g. the implicit cells do not maintain order-invariance, but valuation-invariance).

4.5 Example

As significant portions of the algorithm are taken from the cylindrical algebraic covering method, we again refer to [3] for more intuition of unsatisfiable coverings. In this example, we illustrate how both satisfiable and unsatisfiable cells are characterized for an existentially quantified variable and how coverings of satisfying cells are computed for a universally quantified variable. We consider the following formula with constraints c_1, c_2 and c_3 that are depicted in Fig. 1a:

$$\forall x_1.\ \exists x_2.\ c_1 : x_2 > 3.5-2(x_1-4)^2 \wedge c_2 : (x_1-2)^2+(x_2-2)^2-1 > 0 \wedge c_3 : x_2 < 3+0.25(x_1-4)^2$$

We start with the first variable being universally quantified:

`forall` $(s = ())$ We start covering the real line with satisfiable intervals by sampling values for x_1 (Lines 1 and 2 of Algorithm 4). We then sample any value outside the excluded intervals (in this case, we can pick any value); for illustrational purposes (as for all samples in this example), we choose 2 (Lines 3 of Algorithm 4). As $\overline{\varphi}$ does not evaluate to a value yet, we call the algorithm with the current partial sample to handle the next variable (Lines 9 of Algorithm 4).

`exists(`$s = (2)$`)` We start covering the real line with unsatisfiable intervals (Lines 1 and 2 of Algorithm 3). We sample $x_2 = 3.5$ (Lines 3 of Algorithm 3) and find a satisfying sample. Now, we generalize to the feasible interval around (2, 3.5) as depicted in Fig. 1b, which is bounded from below by c_2 and from above by c_3 (Lines 7 of Algorithm 3). Its projection is the satisfiable interval (1, 3) for x_1 that we return (Lines 11 of Algorithm 3).

We store the received satisfying interval (Lines 11 of Algorithm 4). As there exist samples outside the set of satisfying intervals (Lines 2 of Algorithm 4), we pick the next value 3.2 for x_1 (Lines 3 of Algorithm 4):

`exists(`$s = (3.2)$`)` We sample $x_2 = 2.75$ and find a satisfying sample. We generalize to the feasible interval bounded by c_1 and c_3. Note that in the projection of the feasible interval, we take all constraints into account (as all constraints are part of the implicant), even if they do not have a real root at $x = 3.2$—here, the discriminant of c_2 is added to the projection ensuring that no root of c_2 crosses the feasible interval. The resulting projection is the satisfiable interval $(3, \underline{3.5})$ for x. (The underlined value is an approximation).

Similarly, the received interval is stored, and we proceed with the sample 4 for x_1:

`exists(`$s = (4)$`)` We sample $x_2 = 4$ (Line 3 of Algorithm 3) to obtain the unsatisfiable interval $(3, \infty)$ (Lines 5 of Algorithm 3) which we store in the set of unsatisfying intervals (Lines 14 of Algorithm 3). As this set does not cover the whole real line yet (Lines 2 of Algorithm 3), we sample $x_2 = 2$ (Lines 3 of Algorithm 3) to obtain the unsatisfiable interval $(-\infty, 3.5)$ (Lines 5 of Algorithm 3), which is again stored (Lines 14 of Algorithm 3). The intervals cover the real line for x_2 (Lines 2 of Algorithm 3), as depicted dashed in Fig. 1b. We return the unsatisfiable interval $(\underline{3.5}, \underline{4.5})$ for x_1 which is the projection of the generalization of the covering (Line 15 of Algorithm 3).

As a recursive call returned an unsatisfiable interval, the algorithm terminates here by returning `UNSAT` (Line 14 of Algorithm 4).

5 Quantifier Elimination

From now on, we also allow the input (a formula in prenex normal form) to contain parameters.

To extend the method for quantifier elimination, we could follow a NuCAD [14] like approach: we could "guess" a sample point for all parameters at once, check the satisfiability of the formula using the method above and construct a cell around the sample point. We would iterate this by guessing sample points outside the already constructed cells until no such sample points exist. Finally, we would obtain a list of cells which are either satisfying or unsatisfying.

We propose an alternative approach in Algorithms 9 and 10 which builds upon the cylindrical algebraic covering method in order to obtain a tree-like description of cylindrically arranged cells which the parameter space. This allows for potentially smaller solution formulas and more intuitive reasoning over its structure, as discussed in [2].

The idea is to consider the parameters first, and treat them similarly to existentially quantified variables with a few differences: Instead of returning as soon as we find a satisfiable cell, we collect both satisfiable and unsatisfiable cells until the whole real line

Algorithm 9: `user_call_qe()`

Data : Global prefix $Q_{k+1}x_{k+1}\cdots Q_n x_n$ and matrix $\overline{\varphi}$.
Output : A solution formula for the parameters of $Q_{k+1}x_{k+1}\cdots Q_n x_n.\overline{\varphi}$.

1 **if** $k = 0$ **then**
2 $(f, O) :=$ `recurse(())` `// Algorithm 2`
3 **if** $f =$ `SAT` **then return** `True`
4 **else return** `False`
5 **else**
6 $(\psi, C) :=$ `parameter(())` `// Algorithm 10`
7 **return** ψ

Algorithm 10: `parameter(`s`)`

Data : Global prefix $Q_{k+1}x_{k+1}\cdots Q_n x_n$ and matrix $\overline{\varphi}$.
Input : Sample point $s = (s_1, \ldots, s_{i-1}) \in \mathbb{R}^{i-1}$.
Output : (ψ, C) where ψ characterizes all satisfying cells over s within $s \times C.I$.

1 $\mathbb{I} = \emptyset$
2 $\psi :=$ `False`
3 **while** $\bigcup_{C\in\mathbb{I}} C.I \neq \mathbb{R}$ **do**
4 $s_i :=$ `sample_outside(`$\mathbb{I}$`)`
5 **if** $\overline{\varphi}[s \times s_i] =$ `False` **then**
6 $(T, O) :=$ `(False, get_enclosing_cell(`$s \times s_i$`))` `// Algorithm 6`
7 **else if** $\overline{\varphi}[s \times s_i] =$ `True` **then**
8 $(T, O) :=$ `(True, get_enclosing_cell(`$s \times s_i$`))` `// Algorithm 6`
9 **else if** $i < k$ **then**
10 $(T, O) :=$ `parameter(`$s \times s_i$`)` `// recursive call`
11 **else** it holds $k \leq i < n$
12 $(f, O) :=$ `recurse(`$s \times s_i$`)` `// Algorithm 2, recursive call`
13 **if** $f =$ `SAT` **then** $T :=$ `True`
14 **else** $T :=$ `False`
15 $\mathbb{I} := \mathbb{I} \cup \{O\}$
16 $\psi := \psi \vee$ `(indexed_root_formula(`O`)` $\wedge T$)
17 $C :=$ `characterize_covering(`$s, \mathbb{I}$`)` `// Algorithm 8`
18 **return** (ψ, C)

is covered by them; analogously to Algorithms 3 and 4, we compute a generalization of this covering, this time it consists of satisfiable and unsatisfiable cells instead of only satisfiable or only unsatisfiable ones. These "mixed" coverings are required to ensure that all satisfiable cells of the parameter space are enumerated. Simultaneously, a symbolic description of the satisfiable cells in the parameters is constructed as a formula in Line 16 of Algorithm 10.

For the latter, we employ the concept of *indexed root expressions* [16]:

Definition 5.1 *(Indexed Root Expression)* Let $p \in \mathbb{Q}[x_1, \ldots, x_i]$ and $j \in \mathbb{N}_{>0}$. An *indexed root expression* is a function $\text{root}_{p,j} : \mathbb{R}^{i-1} \to \mathbb{R} \cup \{\text{undefined}\}$; for all $r \in \mathbb{R}^{i-1}$, $\text{root}_{p,j}(r)$ is the j-th real root of the univariate polynomial $p(r, x_i) \in \mathbb{Q}[x_i]$ (or undefined if this root does not exist).

We use constraints over indexed root expressions to describe intervals symbolically:

Definition 5.2 *(Indexed Root Formula)* Let C be an implicit cell in main variable x_i. The set $\Xi_l(C) = \{\text{root}_{p,j} \mid p \in C.P \wedge \text{root}_{p,j}(s_1, \ldots, s_{i-1}) = C.I.l\}$ contains all indexed root expressions bounding the i-th component of C from below, and the set $\Xi_u(C) = \{\text{root}_{p,j} \mid p \in C.P \wedge \text{root}_{p,j}(s_1, \ldots, s_{i-1}) = C.I.u\}$ contains all indexed root expressions

bounding the i-th component of C from above. The *indexed root formula of* C is the formula $\texttt{indexed\_root\_formula}(C) = \bigwedge_{\xi \in \Xi_l(C)} \xi < x_i \wedge \bigwedge_{\xi \in \Xi_u(C)} x_i < \xi$.

Example Consider the polynomials $P = \{x_2 + 1, x_1^2 + x_2^2 - 2, x_1 - 1\}$, the sample point $s = (0, 0)$, and the implicit cell $C = (P, s, (-1, \sqrt{2}))$. Then the indexed root formula of C is $\text{root}_{x_2+1,1} < x_2 \wedge x_2 < \text{root}_{x_1^2+x_2^2-2,2}$.

5.1 Simplification of Results

The above construction of the formula describing the resulting parameter space is naive, as there are low-hanging fruits to simplify these results further. To do so, we store the parameter space in a tree-like *covering data structure* $(\psi_{\text{interval}}, L)$ where ψ_{interval} is an indexed root formula describing some interval symbolically and the *label* L is equal to `True`, `False` (we call those *leafs*) or is a list of covering data structures (we call those *inner nodes*) sorted as in `compute_cover`. We do not only store the satisfying cells, but also the unsatisfying ones (indicated using the first two values for L).

We then simplify by traversing the data structure depth-first:

1. We merge neighboring children T_1, T_2 (where T_1 is sorted before T_2) if both of them do not have children and are labelled both `True` or both `False`. The resulting node T is labelled with the common label, and the interval formula is built from the lower bounds of T_1 and the upper bounds of T_2. We iterate until we cannot merge any more.
2. If a node does have a single child which does not have children itself and is labelled either `True` or `False`, we erase this child, and take over its label to the current node.

Note that by the merging described in the first step, we remove unnecessary atoms from the output formula: The two corresponding cells are neighbors in the same cylinder and thus overlap in the whole cylinder, allowing us to remove the overlapping bounds. In the second step, the single child corresponds to a cell without bounds, thus the value of the corresponding variable does not affect the output of the algorithm.

The construction of a formula from this data structure is straight-forward by iterating through the tree and encoding the satisfiable cells. However, we propose another optimization during this process, based on the observation that for some subtrees, there are fewer unsatisfying cells than satisfying ones. In those cases, we want to encode the unsatisfying cells instead of the satisfying ones. We encode a leaf in the obvious way, and each inner node as follows:

1. We count the number of leaf children labelled with `True` and `False` respectively.
2. If the first number is smaller, we encode all leaf children labelled with `True` and recursively encode all inner children.
3. If the latter number is smaller, we encode all leaf children labelled with `False` and recursively encode all inner children. For the latter, we pass a flag indicating that the encoding result should describe unsatisfying cells.
4. We build the disjunction of all cells. Depending on which of the two cases we entered and the received flag, we negate this disjunction. Afterwards, we build the conjunction of the (possibly negated) disjunction with the indexed root formula of the current node.

Last but not least, we remark that the techniques for CAD-based quantifier elimination could also be adapted for further simplifying the output; in particular the *SimpleSF*

algorithm described in [16, Section 5.2] is promising. However, to apply it, we would possibly need to refine the given covering to a decomposition first, which would require computational and technical effort.

5.2 Elimination of Indexed Root Expressions

While indexed root expressions are outside the language admitted by standard non-linear real arithmetic, equivalent "pure" non-linear real arithmetic formulas can be constructed with some effort. In the following, we discuss some possibilities for doing so, based on common techniques also used in other algorithms. They are not specific to the CAlC method, but worth noting in this context.

A *sign condition* of a set P of polynomials in $x_1, \ldots, x_i$ assigns a sign (positive, negative or zero) to each polynomial; a point in $\mathbb{R}^i$ *satisfies* a sign condition if the evaluation of each polynomial at s corresponds to its assigned sign. *Thom's lemma* (see [40, pp. 325–326]) states that the maximal set satisfying some sign condition of a univariate polynomial and all its derivatives is either empty or connected. In other words, we can describe intervals using constraints whose defining polynomials are derivatives of some polynomial.

Given a formula φ (containing indexed root expressions) that describes a single cell (which can be extracted efficiently from the covering data structure) and a sample $s \in \mathbb{R}^i$ that lies in the cell, we can eliminate all indexed root expressions using a generalization of Thom's lemma to multivariate polynomials obtained by viewing them as univariate polynomials in x_i with polynomial coefficients. We may use this generalization in different ways, resulting in the following three different approaches:

- Let P be the set of defining polynomials of φ, and $P' = \{\frac{\partial^k}{\partial x_i^k} p(x_1, \ldots, x_i) \mid p \in P, i = \text{level}(p), k = 0, \ldots, \deg_{x_i}(p)\}$ be the set containing all $p \in P$ and all of their partial derivatives $\frac{\partial^k}{\partial x_i^k} p$ with respect to the respective main variable x_i. We compute a set of sign conditions on the polynomials in P' such that the union of their described cells is equal to the cell described by φ. We can do so by starting from s, storing the sign condition that satisfies s; and then move to an adjacent cell by changing a single sign in the sign condition. If there is a sample s' that satisfies both φ and the adapted sign condition, we store the new sign condition. Otherwise we go back. We iterate this until this yields no more sign conditions (when starting from any sign condition in the set); then, all adjacent cells do not satisfy φ. Finally, we encode all obtained sign conditions by polynomial constraints.
- We compute the *augmented projection* of P, that is the closure of the CAD projection and partial derivatives w.r.t. the respective main variable, as described in [20, p. 144]. This projection yields a CAD that is *projection definable* [16, Definition 3], i.e., the CAD can be encoded using sign-conditions on the projection factors. The difference to the previous approach is the following: all derivatives are delineable in the respective lower-level cells, meaning that the resulting formula is cylindrically arranged. Thus, the resulting formula might require fewer atoms by encoding the cells in a tree-like manner.
- Using techniques from [16], we can optimize the previous approach by reducing the amount of derivatives that are added to the projection set - not all derivatives are necessary for obtaining a projection-definable CAD.

The computation of derivatives required in the above approaches might lead to large output formulas and additional expensive CAD computations. It is thus desirable to reduce this effort. This is achieved by encoding each cell separately, but encoding all cells in combination in order to eliminate redundancies of sign conditions on the derivatives (effectively reducing the amount of required derivatives). The work in [16] proposes efficient algorithms for this task; we thus could feed the output formula into these algorithms (either all at once or incrementally as in [16, Section 6]).

Certainly, obtaining a small output formula without indexed root expression requires additional computational cost and implementation effort. Moreover, formulas containing indexed root expressions allow for smaller encodings.

6 Implicant Calculation

The presented algorithms rely on `implicant_polynomials` to compute an implicant that generalizes the reason why a formula simplifies to either `True` or `False` at a given sample point, as defined in Definition 3.3. We propose different variants for computing such an implicant. In all of these variants, the choice of the implicant is generally not unique. We always define the set of all implicants that can be computed in the described way. In an implementation, we could either collect all possible implicants and decide afterwards which one to take, or, to reduce the computational effort, compute a single good implicant.

We remark that the algorithms presented in this section are kept simple for pedagogical and experimental purposes. We assume that the formula is in prenex normal form and its matrix φ is in negation normal form.

6.1 Evaluation Only

We omit Boolean reasoning and simply evaluate the formula using the given sample point. Let $s \in \mathbb{R}^i$ such that $\varphi[s] = \texttt{False}$. We compute the set of implicants recursively:

$$\text{implicants}_s(c) = \{\} \text{ if } c \text{ is a constraint and } c(s) \neq \texttt{False}$$
$$\text{implicants}_s(c) = \{\{\neg c\}\} \text{ if } c \text{ is a constraint and } c(s) = \texttt{False}$$
$$\text{implicants}_s(\psi_1 \wedge \psi_2) = \text{implicants}_s(\psi_1) \cup \text{implicants}_s(\psi_2)$$
$$\text{implicants}_s(\psi_1 \vee \psi_2) = \text{implicants}_s(\psi_1) \times \text{implicants}_s(\psi_2)$$

The case $\varphi[s] = \texttt{True}$ is analogous, as we basically switch the cases $\psi_1 \wedge \psi_2$ and $\psi_1 \vee \psi_2$. We interpret the resulting sets of constraints as conjunction.

6.2 Boolean Propagation

The above approach misses important Boolean information:

Example Consider $\varphi_1 = (x < 0 \vee y = 0) \wedge (x < 1 \vee y \neq 0)$, which is unsatisfiable over $x = 2$, but our partial evaluation does not detect that the resulting formula is logically equivalent to `False`: As the set of implicants for $y = 0$ and $y \neq 0$ is empty, the sets of implicants of all subformulas and the formula are empty as well.

We thus incorporate Boolean reasoning that resembles the Boolean propagation implemented within SAT solvers. Consider Algorithms 11 and 13. Let $s \in \mathbb{R}^i$. If `implicants(`φ`, (`φ`), s)` (`implicants(`φ`, (`$\neg\varphi$`), s)`) is non-empty, then $\varphi[s] \equiv$

`False` ($\varphi[s] \equiv$ `True`), and the algorithm already computes all possible implicants. The algorithm takes a formula φ, a sequence of decisions D (its entries are subformulas of φ or their negation) and a sample point s as input. The algorithm maintains for each subformula two sets $T(\psi)$ and $F(\psi)$ of *reasons* (which are sets of formulas maintained continuously through the algorithm) implying the subformula evaluates to `True` or `False`, respectively, given the decisions in D and the sample s. For every decision $d \in D$, its set of reasons $T(d)$ contains only the empty set, which represents an "unconditional" reason. Every subformula ψ and its negation $\neg\psi$ share the same sets (e.g. $T(\psi)$ is the same as $F(\neg\psi)$). Note that the `evaluate` subroutine corresponds to the evaluation only approach. The algorithm yields an implicant whenever it finds a conflict, that is, some subformula has both reasons to evaluate to `True` and to `False`, contradicting the assumption that φ (respectively, $\neg\varphi$) holds. We note that this scheme can be extended to arbitrary Boolean operators such as exclusive-or and negation.

Example Consider φ_1 and the sample $x = 2$ from the previous example again. Using the sample, we conclude $F(x < 0) = \{\{x < 0\}\}$ and $F(x < 1) = \{\{x < 1\}\}$. Additionally, we decide $T(\varphi_1) = \{\emptyset\}$ and propagate $T(x < 0 \vee y = 0) = \{\emptyset\}$ and $T(x < 1 \vee y \neq 0) = \{\emptyset\}$. By propagation via the disjunctions, we conclude that $T(y = 0) = F(y \neq 0) = \{\{x < 1\}\}$ and $T(y \neq 0) = F(y = 0) = \{\{x < 1\}\}$. Thus, $T(y = 0) \times F(y = 0) = \{\{x < 0, x < 1\}\}$ is the set of implicants.

Algorithm 11: `implicants(`φ, D, s`)`

```
1  foreach subformula ψ of φ do T(ψ) := ∅
2  foreach subformula ψ of φ do F(ψ) := ∅
3  foreach d in D do T(d) := {∅}
4  foreach atom c in φ do
5      if c(s) = True then T(c) := {{c}}
6      else if c(s) = False then F(c) := {{¬c}}
7  while T or F changed do
8      foreach subformula ψ of φ do
9          evaluate(ψ)
10         propagate(ψ)
11 return ⋃_{ψ∈φ} T(ψ) × F(ψ)
```

Algorithm 12: `explore(`φ, D, s`)`

```
1 I_D := implicants(φ, D, s)
2 if I_D ≠ ∅ then return I_D
3 choose subformula ψ of φ s.t. T(ψ) = F(ψ) = ∅ at the end of the implicants call
4 if no such ψ exists then return ∅
5 I_(D,ψ) := explore(φ, (D, ψ), s)
6 I_(D,¬ψ) := explore(φ, (D, ¬ψ), s)
7 return I_(D,ψ) × I_(D,¬ψ)
```

Algorithm 13: `evaluate` and `propagate`

```
1  def evaluate(ψ1 ∧ ψ2):
2      T(ψ1 ∧ ψ2) ∪= T(ψ1) × T(ψ2)
3      F(ψ1 ∧ ψ2) ∪= F(ψ1) ∪ F(ψ2)
4  def evaluate(ψ1 ∨ ψ2):
5      T(ψ1 ∨ ψ2) ∪= T(ψ1) ∪ T(ψ2)
6      F(ψ1 ∨ ψ2) ∪= F(ψ1) × F(ψ2)
7  def propagate(ψ1 ∧ ψ2):
8      T(ψ1) ∪= T(ψ1 ∧ ψ2)
9      T(ψ2) ∪= T(ψ1 ∧ ψ2)
10     F(ψ1) ∪= F(ψ1 ∧ ψ2) × T(ψ2)
11     F(ψ2) ∪= F(ψ1 ∧ ψ2) × T(ψ1)
12 def propagate(ψ1 ∨ ψ2):
13     T(ψ1) ∪= T(ψ1 ∨ ψ2) × F(ψ2)
14     T(ψ2) ∪= T(ψ1 ∨ ψ2) × F(ψ1)
15     F(ψ1) ∪= F(ψ1 ∨ ψ2)
16     F(ψ2) ∪= F(ψ1 ∨ ψ2)
```

6.3 Boolean Exploration

The previous approach is able to detect some Boolean conflicts by propagation. However, there are formulas which are not satisfiable by the Boolean structure already, which can only be detected by exploration:

Example Consider $\varphi_2 = ((z = 0 \vee \varphi_1) \wedge (z \neq 0 \vee \varphi_1))$ and the sample $x = 2$. This formula is clearly unsatisfiable. However, although we would evaluate the literals ($F(x < 0) = \{\{x < 0\}\}$ and $F(x < 1) = \{\{x < 1\}\}$) and decide $T(\varphi_2) = \{\emptyset\}$, we would not be able to propagate further. We thus need to check the cases $z = 0$ and $z \neq 0$.

Consider Algorithm 12. Let $s \in \mathbb{R}^i$. If `explore(`φ`, (`φ`), `s`)` (`explore(`φ`, (`$\neg\varphi$`), `s`)`) is non-empty, then $\varphi[s] \equiv$ `False` ($\varphi[s] \equiv$ `True`), and the algorithm again computes all possible implicants. The algorithm naively assumes formulas to evaluate to `True` and `False`, calls recursively, and if both choices lead to a conflict, then it combines the obtained implicants. We note that also a partial exploration of the search space is possible.

Example Assume we call `explore` on φ_2 from the previous example, $D = (\varphi_2)$ and $s = (2)$ (i.e. $x = 2$). The implicants call would return the empty set, we thus choose a subformula, for example $\psi := z = 0$ thus recursively call `explore` on φ_2 and $D = (\varphi, z = 0)$ and on φ_2 and $D = (\varphi, z \neq 0)$. The first call would set the reasons of $z = 0$ to $\emptyset$, propagate these reasons and detect an immediate conflict from this decision, thus return the implicant $\{x < 0, x < 1\}$; similarly, the second call would return $\{x < 0, x < 1\}$. Thus the overall call would return $\{x < 0, x < 1\} \times \{x < 0, x < 1\} = \{x < 0, x < 1\}$ as implicant.

6.4 Inprocessing

The implicant could be further simplified using preprocessing techniques, such as Gröbner bases ([28] predicts whether preprocessing using Gröbner bases benefits a CAD computation) or techniques described in [19]. We do not need bookkeeping of relations of input and output constraints, as we do not need to return infeasible subset as in the CDCL(T) framework.

7 Exploiting the Quantifier Structure

So far, we assumed the input to be in prenex normal form. In the following, we lift this restriction to better exploit the quantifier structure by switching the quantifier order and solving independent subformulas separately. Potentially, we can rule out parts of the formula as being irrelevant in the current branch. To some degree, this is already facilitated by choosing implicants; however, implicants only consider the Boolean structure and completely ignore the quantifiers!

Example Consider the formula $\varphi = \forall x \forall y.\ (\psi_1(x,y) \wedge \psi_2(x,y))$, which is logically equivalent to $\forall x \forall y.\ \psi_1(x,y) \wedge \forall x \forall y.\ \psi_2(x,y)$. We can check $\forall x \forall y.\ \psi_1(x,y)$ and $\forall x \forall y.\ \psi_2(x,y)$ separately and combine the results accordingly. For each subformula, we can even choose a different variable ordering.

7.1 Input Transformation

We proceed as follows: Assume that we transformed the input formula to negation normal form, that is we pushed the negations into the formula using the double negation rule and De Morgan's rules such that only the atoms occur negatively in the formula. Afterwards, we push the quantifiers as far as possible into the formula using the following rules:

$$QxQy.\ \varphi(x,y) \equiv QyQx.\ \varphi(x,y) \text{ for } Q \in \{\exists, \forall\} \qquad \text{(Swapping Quantifiers)}$$

$$Qx.\ \varphi \equiv \varphi \text{ for } Q \in \{\exists, \forall\} \qquad \text{(Null Quantification)}$$

$$\forall x.\ (\varphi(x) \wedge \psi(x)) \equiv \forall x.\ \varphi(x) \wedge \forall x.\ \psi(x) \qquad \text{(Distribution over Conjunction)}$$

$$\exists x.\ (\varphi(x) \vee \psi(x)) \equiv \exists x.\ \varphi(x) \vee \exists x.\ \psi(x) \qquad \text{(Distribution over Disjunction)}$$

$$Qx.\ (\varphi(x) \circ \psi) \equiv (Qx.\ \varphi(x)) \circ \psi \text{ for } Q \in \{\exists, \forall\},\ \circ \in \{\wedge, \vee\}$$
(Prenex Law for {Con,Dis}junction)

Note that we can apply the Equation (Swapping Quantifiers) for formulas in prenex normal form as well. In this case, it is rather clear how to make use of this rule - we just choose a typical variable ordering that works well for the CAD. In the general case, the order in which we apply Equation (Swapping Quantifiers) and the other rules might affect the result. Thus, the variable ordering has an influence on how we can distribute the quantifiers.

Example Consider the formula $\varphi = \forall x \forall y.\ (\psi_1(x,y) \wedge \psi_2(y))$. When keeping the variable ordering, we can transform the formula to $\forall x.\ (\forall y.\ \psi_1(x,y) \wedge \forall y.\ \psi_2(y))$, although $\psi_2(y)$ does not depend on x! When switching the quantifiers, we can transform the formula to $\forall y \forall x.\ \psi_1(x,y) \wedge \forall y.\ \psi_2(y)$.

We could transform the formula iteratively after each assignment of a variable in the covering algorithm by plugging it in to all constraints and propagating the truth values:

Example Consider the formula $\varphi = \forall x \forall y.\ [(x > 0 \wedge \psi_1(x,y) \wedge \psi_2(x,y)) \vee (x \leq 1 \wedge \psi_3(x,y))]$. Due to the disjunction, we cannot pull any quantifier into the formula.

Now assume we picked $x = 2$, thus the second case $\vartheta = (x \leq 1 \wedge \psi_3(x,y))$ of φ simplifies to `False`. An implicant for ϑ w.r.t. 2 is $x > 1$. We use this information to rewrite the formula to

$$\forall x \forall y.\ [(x > 0 \wedge \psi_1(x,y) \wedge \psi_2(x,y)) \vee x > 1]$$

$$\equiv \forall x.\ [(x > 0 \wedge \forall y.\ \psi_1(x, y) \wedge \forall y.\ \psi_2(x, y)) \vee x > 1]$$

which allows us to split the problem into multiple ones.

7.2 Adaption of the Algorithm

We adapt our algorithm to work on general non-linear real arithmetic formulas involving quantifiers and to explore independent subformulas separately, as depicted in Algorithms 14 to 16. First note that the formula φ is now a parameter of all algorithms called recursively. The `recurse` algorithm (Algorithm 14) now handles the cases where the formula is detected to be either equivalent to `True` (Line 3) or `False` (Line 1) after plugging in s. We note that `implicant_polynomials`, which is indirectly called in Lines 2 and 4, needs to be extended for quantified formulas as input, still computing a quantifier-free implicant. We modify Definition 3.3 as follows:

Algorithm 14: `recurse(`φ`, s)`

Input : NRA formula φ, sample point $s = (s_1, \ldots, s_{i-1}) \in \mathbb{R}^{i-1}$.
Output : (SAT, C) or (UNSAT, C) where $s \times C.I$ can or can not be extended to a model for any $s_i \in C.I$. In both cases, C describes how s can be generalized.

1 **if** $\varphi[s] \equiv$ `False` **then**
2 | **return** (UNSAT, `get_enclosing_cell(`φ, s`)`) `// Algorithm 6`
3 **else if** $\varphi[s] \equiv$ `True` **then**
4 | **return** (SAT, `get_enclosing_cell(`φ, s`)`) `// Algorithm 6`
5 **else** it holds $i < n$
6 | **transform** φ based on s
7 | **if** $\varphi = \exists x_{i+1}.\ \psi$ **then**
8 | | **return** `exists(`ψ, s`)` `// Algorithm 15`
9 | **else if** $\varphi = \forall x_{i+1}.\ \psi$ **then**
10 | | **return** `forall(`ψ, s`)` `// Algorithm 16`
11 | **else if** $\varphi = \bigvee_{j=1,\ldots,k} \psi_j$ **then**
12 | | $\mathbb{I}_{\text{unsat}} := \emptyset$
13 | | **sort** $\psi_1, \ldots, \psi_k$ according to some heuristic
14 | | **foreach** $j = 1, \ldots, k$ **do**
15 | | | $(f, O) :=$ `recurse(`ψ_j, s`)`
16 | | | **if** $f =$ SAT **then return** (SAT, O)
17 | | | **else if** $f =$ UNSAT **then** $\mathbb{I}_{\text{unsat}} := \mathbb{I}_{\text{unsat}} \cup \{O\}$
18 | | **return** (UNSAT, $\bigcap_{C \in \mathbb{I}_{\text{unsat}}} C$)
19 | **else if** $\varphi = \bigwedge_{j=1,\ldots,k} \psi_j$ **then**
20 | | $\mathbb{I}_{\text{sat}} := \emptyset$
21 | | **sort** $\psi_1, \ldots, \psi_k$ according to some heuristic
22 | | **foreach** $j = 1, \ldots, k$ **do**
23 | | | $(f, O) :=$ `recurse(`ψ_j, s`)`
24 | | | **if** $f =$ UNSAT **then return** (UNSAT, O)
25 | | | **else if** $f =$ SAT **then** $\mathbb{I}_{\text{sat}} := \mathbb{I}_{\text{sat}} \cup \{O\}$
26 | | **return** (SAT, $\bigcap_{C \in \mathbb{I}_{\text{sat}}} C$)

Definition 7.1 *(Implicant)* Let $s \in \mathbb{R}^i$ be a (partial) sample point and φ be a formula with free variables $x_1, \ldots, x_i$. If $\varphi[s] \equiv$ `True`, then the quantifier-free formula ψ is an *implicant of φ with respect to s* if

$$\psi[s] = \texttt{True} \wedge (\psi \Rightarrow \varphi)$$

Algorithm 15: `exists(φ, s)`

Input : NRA formula φ, sample point $s = (s_1, \ldots, s_{i-1}) \in \mathbb{R}^{i-1}$.
Output : see Algorithm 14

```
1  I_unsat := ∅
2  while ⋃_{C∈I_unsat} C.I ≠ ℝ do
3  |  s_i := sample_outside(I_unsat)
4  |  (f, O) := recurse(φ, s × s_i)
5  |  if f = SAT then
6  |  |  C := characterize_cell(s, O)
7  |  |  return (SAT, C)
8  |  else if f = UNSAT then
9  |  |  I_unsat := I_unsat ∪ {O}
10 C := characterize_covering(s, I_unsat)
11 return (UNSAT, C)
```

Algorithm 16: `forall(φ, s)`

Input : NRA formula φ, sample point $s = (s_1, \ldots, s_{i-1}) \in \mathbb{R}^{i-1}$.
Output : see Algorithm 14

```
1  I_sat := ∅
2  while ⋃_{C∈I_sat} C.I ≠ ℝ do
3  |  s_i := sample_outside(I_sat)
4  |  (f, O) := recurse(φ, s × s_i)
5  |  if f = SAT then
6  |  |  I_sat := I_sat ∪ {O}
7  |  else if f = UNSAT then
8  |  |  C := characterize_cell(s, O)
9  |  |  return (UNSAT, C)
10 C := characterize_covering(s, I_sat)
11 return (SAT, C)
```

and the constraints of ψ are of level at most i and contained in φ. Otherwise, if $\varphi[s] \equiv \texttt{False}$, then the quantifier-free formula ψ is an *implicant of φ with respect to s* if

$$\psi[s] = \texttt{True} \wedge (\psi \Rightarrow \neg\varphi)$$

and the constraints of ψ are of level at most i and contained in φ.

Example Consider the formula $\varphi = x_1 < 0 \vee \forall x_2.\ (x_2 > 0 \rightarrow x_1 > 2)$ and assume the sample (1), then $\varphi[(1)] = \texttt{False}$ and $0 \leq x_1 \wedge x_1 \leq 2$ is an implicant for φ w.r.t. (1).

If neither of the first two cases hold ((Lines 1 and 3), we transform the formula based on the current sample point in Line 6 to facilitate splitting as described at the end of Sect. 7.1. Then, we do a case distinction on the formula's structure: If the formula is a quantified formula (Lines 7 and 9), we call the algorithm handling the respective quantifier. If the formula is a conjunction or disjunction (Lines 11 and 19), we call Algorithm 14 recursively on the individual subformulas. Analogously to the exists (forall) case, for disjunctions (conjunctions), we return early once one of the recursive calls returns a satisfying (unsatisfying) cell; otherwise, we collect all unsatisfying (satisfying) cells and build their intersection.

Such an *intersection C' of implicit cells* $C_1, \ldots, C_k$ where $C_1.s = \ldots = C_k.s$ is defined such that $C'.P = \cup_{j=1,\ldots,k} C_j.P$, $C.s = C_1.s$, $C'.I = \cap_{j=1,\ldots,k} C_j.I$. Note that, by definition, the intersection is non-empty, as the implicit cells share the same sample point which is also contained in the intersection. Further, the algorithm only applies the intersection on implicit cells with the same sample point.

We emphasize that the splitting mechanism in Lines 11 and 19 overlaps with the implicant calculation. In an efficient implementation, this would be interleaved with the calculation of implicants, possibly considering the whole Boolean structure of φ at once to traverse the search tree "non-chronologically". We could compute valid combinations of recursive calls and choose to "best" combination according to some metric.

Algorithms 15 and 16 depict the new algorithms for handling existentially and universally quantified variables. Compared to Algorithms 3 and 4, the calls to `get_enclosing_cell` are moved to `recurse`. Analogously to Sect. 4, we could extend this approach for parameters in quantifier elimination problems; as this is straight-forward, we omit it here.

8 Proof System

This section replaces the CAD projection algorithms given above by the proof system introduced in [42]. This proof system changes the view of "computing projections of polynomials in one variable less" as in classical CAD formulations to "computing properties that a lower-level cell S needs to fulfil such that we can describe a cell in the cylinder $S \times \mathbb{R}$"; these properties ultimately prove that a polynomial is sign-invariant on a cell.

The motivation is twofold: Firstly, the proof system allows for more efficient projections, as its modular formulation can consider many fine-grained optimizations while keeping the algorithmic aspects clean. Secondly, the proof system might be a step towards a proof-producing procedure for non-linear real arithmetic formulas, i.e., generating (formal) proofs that can be mechanically verified: While the CAlC method would define a high level proof strategy (as motivated in [2,4] and mentioned in [35]), the proof system would provide a fine(r)-grained layer (as discussed in [42, Section 8.1]).

For the following section, we assume the reader being familiar with the proof system in [42]. We briefly recall the most important definitions before presenting an extension for coverings.

8.1 Preliminaries

The work in [42] introduces a proof system for the *single cell construction* [18] algorithm, which generalizes unsatisfiable sample points to unsatisfiable cells in MCSAT-based solvers [32]. The input is a set $P \subseteq \mathbb{Q}[x_1, \dots, x_n]$ of polynomials and a sample point $s \in \mathbb{R}^n$, and we aim to find a description of a cell $R \subseteq \mathbb{R}^n$ such that $s \in R$ and each $p \in P$ is sign-invariant in R. The algorithm iteratively computes *symbolic intervals* $\mathtt{I}_n, \dots, \mathtt{I}_1$ for the variables $x_n, \dots, x_1$ such that for each $i = 1, \dots, n$, the bounds of $\mathtt{I}_i$ depend on $x_1, \dots, x_{i-1}$.

Definition 8.1 (Symbolic Interval [42]) A *symbolic interval of level i* is either a tuple (section, b) where b is an indexed root expression with domain $\mathbb{R}^{i-1}$, or (sector, l, u) where each of l and u is either $-\infty/\infty$ respectively or an indexed root expression with domain $\mathbb{R}^{i-1}$. Intervals of the former represent *sections* $\{(r, b(r)) \mid r \in R\}$ (where $R \subseteq \mathbb{R}^{i-1}$ such that b is defined), intervals of the latter represent *sectors* $\{(r, r') \mid r \in R, r' \in (l(r), u(r))\}$ (where $R \subseteq \mathbb{R}^{i-1}$ such that l and u are defined).

To go into more detail, assume we construct the interval $\mathtt{I}_n$. To maintain sign- resp. order-invariance of P in the resulting cell R, we first choose $\mathtt{I}_n$ such that its boundaries are defined by roots of P and s_n is either equal to $b(s_{[n-1]})$ or contained in the interval $(l(s_{[n-1]}), u(s_{[n-1]}))$. We now need to compute the lower-level intervals such that the described underlying cell $R{\downarrow_{[n-1]}}$ is small enough such that $\mathtt{I}_n$ describes a sign-invariant interval for P above each point $r \in R{\downarrow_{[n-1]}}$. To do so, we need to ensure that no root $\xi : \mathbb{R}^{n-1} \to \mathbb{R}$ of P crosses a boundary of $\mathtt{I}_n$, meaning that $\xi(r) \leq l(r)$ or $u(r) \leq \xi(r)$ for all $r \in R{\downarrow_{[n-1]}}$. We need to maintain a certain *ordering* on the roots:

Definition 8.2 (Indexed Root Ordering [42]) An *indexed root ordering of level i* is a relation $\preceq$ on a set Ξ of indexed root expressions with domain $\mathbb{R}^i$ such that its reflexive and transitive closure $\preceq^t$ is a partial order on Ξ. We say it *matches* some $s \in \mathbb{R}^{i-1}$ if all Ξ are defined at s and $\xi \preceq \xi'$ implies $\xi(s) \leq \xi'(s)$ for all $\xi, \xi' \in \Xi$.

We first determine an ordering $\preceq$ on all the roots such that $\xi \preceq^t l$ or $u \preceq^t \xi$ for all roots ξ of P that we can "see" at $s_{[i-1]}$ (where $\preceq^t$ is the transitive and reflexive closure of $\preceq$). We use CAD projection tools to maintain that this ordering is maintained on $R{\downarrow_{[n-1]}}$, i.e. we compute a set of polynomials P' in $n-1$ variables whose sign-invariance guarantees this property. We iteratively apply the described procedure on the lower levels, until intervals for all variables are computed.

Observe that above, we allow for some flexibility in the choice of the indexed root ordering by exploiting transitivity. Throughout our procedure, there are more such choices possible, as well as many optimizations in the CAD projection theory which are only applicable in certain cases. A proof system keeps the algorithm maintainable while exploiting these cases: We define *properties of level i* which are functions $q : \{R \mid R \subseteq \mathbb{R}^i\} \to \{0, 1\}$ for $i = 1, \ldots, n$. Each proof rule has a single property as consequent; its antecedents are "smaller" properties (according to some ordering in the properties) and side conditions which "enable" the proof rule.

Given $R \subseteq \mathbb{R}^i$, we define properties $\mathit{sample}(s)(R) = 1$ iff $s \in R$, $\mathit{connected}(i)(R) = 1$ iff R is connected, $\mathit{ir\_ord}(\preceq, s)(R) = 1$ iff $\preceq$ matches s and R maintains the ordering $\preceq$, $\mathit{an\_del}(p)(R) = 1$ iff the polynomial p is analytically delineable on some connected superset of R, and $\mathit{repr}(\mathtt{I}, s)(R) = 1$ iff $\mathtt{I}$ is defined at s and the i-th dimension of R is described by $\mathtt{I}$. For more details, we refer to [42].

8.2 Proof Rules for Coverings

The CAlC algorithm shares similarities with the single cell construction algorithm. In fact, the computation of a single symbolic interval corresponds to Algorithm 7. In the following, we will extend the proof system to also cover Algorithm 8.

We start by defining a property that holds iff a set of symbolic intervals covers the whole real line if we substitute a sample point s:

Definition 8.3 Let $i \in \mathbb{N}_{>0}$, $R \subseteq \mathbb{R}^i$, $s \in \mathbb{R}^{i-1}$, and $\mathtt{C} = (\mathtt{I}_1, \ldots, \mathtt{I}_k)$ be a sequence of symbolic intervals of level i.

The property $\mathit{cov}(\mathtt{C}, s)$ holds on R if and only if for every $R' \subseteq \mathbb{R}^{i+1}$ with $R'{\downarrow_{[i]}} = R$ there exists a $j \in \{1, \ldots, k\}$ such that the property $\mathit{repr}(\mathtt{I}_j, s)$ holds on some superset of R'.

This property can be proven using the following rule. It assumes that the intervals are ordered by their lower bounds and are not redundant as in `compute_covering`. We then use an indexed root ordering which ensures that the bounds of neighboring intervals overlap.

Lemma 8.4 *Let $i \in \mathbb{N}_{>0}$, $R \subseteq \mathbb{R}^i$, $s \in \mathbb{R}^{i-1}$, $C = (\mathtt{I}_1, \ldots, \mathtt{I}_k)$ be a sequence of symbolic intervals of level i, and $\preceq$ be an indexed root ordering of level i.*

Assume that C fulfils the following conditions:

- $\mathtt{I}_1.l = -\infty$ *and* $\mathtt{I}_k.u = \infty$,
- $\mathtt{I}_j.l(s) < \mathtt{I}_{j+1}.l(s) \vee (\mathtt{I}_j.l(s) = \mathtt{I}_{j+1}.l(s) \wedge (\mathtt{I}_j \textit{ section} \wedge \mathtt{I}_{j+1} \textit{ sector}))$ *for* $j = 1, \ldots, k-1$,
- $\mathtt{I}_j.u(s) < \mathtt{I}_{j+1}.u(s) \vee (\mathtt{I}_j.u(s) = \mathtt{I}_{j+1}.u(s) \wedge (\mathtt{I}_j \textit{ sector} \wedge \mathtt{I}_{j+1} \textit{ section}))$ *for* $j = 1, \ldots, k-1$, *and*
- $\mathtt{I}_{j+1}.l(s) < \mathtt{I}_j.u(s) \vee (\mathtt{I}_{j+1}.l(s) = \mathtt{I}_j.u(s) \wedge (\mathtt{I}_{j+1} \textit{ section} \vee \mathtt{I}_j \textit{ section}))$ *for* $j = 1, \ldots, k-1$.

$$\begin{gathered}\textit{Assume that } \preceq \textit{ matches } s, \textit{ and for } j = 1, \dots, k-1 \textit{ it holds } \mathtt{I}_{j+1}.l \preceq^t \mathtt{I}_j.u.\\ \textit{sample}(s)(R),\ \textit{connected}(i)(R),\ \textit{ir\_ord}(\preceq, s)(R),\\ \forall j = 1, \dots, k-1.\ (\textit{an\_del}(\mathtt{I}_j.u.p, s)(R) \wedge \textit{an\_del}(\mathtt{I}_{j+1}.l.p, s)(R)) \ \vdash \textit{cov}(C, s)\end{gathered}$$

Proof (Sketch) We ensure that all polynomials defining the lower and upper bounds of the symbolic intervals are analytically delineable on a connected set which contains the current sample point, that means that all their root functions are well-defined over that set. Further, we maintain an ordering of these root functions which ensures that the symbolic intervals cover the cylinder over that cell. For the latter, first observe that the bullet points encode the same requirements as the output of `compute_covering`, i.e. that the intervals are sorted according to their lower bounds, that they are overlapping and not redundant. To maintain these overlaps over the underlying set, we require that the indexed root ordering fulfils that the lower and upper bound of all neighboring pairs of symbolic intervals remain in that same order.

8.3 Adaption of Algorithms

We adapt the CAlC algorithm as follows: First, instead of representing an implicit cell with a tuple (P, s, I) where P is a set of polynomials that are sign- or order-invariant in the cell, we represent it as a tuple (Q, s, I) where Q is a set of properties that hold in the cell. We adapt Algorithms 6 to 8 to work with the proof system, as given in Algorithms 17 to 19.

Algorithm 17 initializes the set Q with the sign-invariance of the implicant's polynomials and applies some basic rules such as factorization, and computes the interval above the given sample. Algorithm 18 adds connectedness of the constructed cell to the set of properties, as the proof system does not always produce descriptions of connected sets. It then isolates the real roots, determines a symbolic interval, an indexed root ordering, and a set E for which we refer to [42] for details. It then applies all proof rules to the point where the interval above the given sample can be determined. Algorithm 19 computes a sequence of non-redundant, ordered intervals representing a covering, isolates the roots of each implicit cell, and then determines all symbolic intervals, and a single indexed root ordering which both protects each cell individually and ensures that the boundaries of the symbolic intervals overlap. By choosing a single root ordering, we might be able to rule out redundancies in the root orderings and thus obtain a more efficient projection; however, for now, we do not make use of this possibility and compute orderings for each cell separately, and the trivial ordering that maintains the covering afterwards. The algorithm then applies all proof rules to the point where the interval above the given sample can be determined.

9 Experimental Evaluation

9.1 Implementation and Heuristics

Our implementation incorporates all the algorithms in this paper except the elimination of indexed root expressions for quantifier elimination (Sect. 5.2) and the techniques for exploiting the quantifier structure (Sect. 7); we postponed the implementation of the first due to the high effort, and the latter as this would require deeper changes of our data structures. We use McCallum's projection operator, which is technically incomplete.

Algorithm 17: `get_enclosing_cell(`s`)`

Data : Global matrix $\overline{\varphi}$.
Input : Sample point $s \in \mathbb{R}^i$ such that $\overline{\varphi}[s] \equiv$ `False` or $\overline{\varphi}[s] \equiv$ `True`.
Output : A satisfiability-invariant implicit cell C containing s.

$P :=$ `implicant_polynomials(`$\overline{\varphi}, s$`)`
$Q := \{sgn\_inv(p) \mid p \in P\}$
apply proof rules to Q until only properties $sgn\_inv(p)$ where p is irreducible remain
$C :=$ $(Q, s,$ `compute_cell(`$s, \{p \mid sgn\_inv(p) \in Q\}$`))` `// Algorithm 5`
return C

Algorithm 18: `characterize_cell(`$s,\ C$`)`

Input : Sample point $s \in \mathbb{R}^i$ and an implicit cell $C = (\cdot, s \times \cdot, \cdot)$ of level $i+1$.
Output : A satisfiability-invariant implicit cell C containing s.

$(Q, s', \cdot) := C$
$Q := Q \cup \{connected(i+1)\}$
compute set Ξ of symbolic roots of $\{p \mid sgn\_inv(p) \in Q\}$
choose representation $(\mathtt{I}, E, \preceq)$ of Ξ w.r.t. s'
apply proof rules to Q considering $(\mathtt{I}, E, \preceq)$ until only properties $sgn\_inv(p)$ (of level i) where p is irreducible remain
$C :=$ $(Q, s,$ `compute_cell(`$s, \{p \mid sgn\_inv(p) \in Q\}$`))` `// Algorithm 5`
return C

Algorithm 19: `characterize_covering(`$s,\ \mathbb{I}$`)`

Input : Sample point $s \in \mathbb{R}^i$ and a set $\mathbb{I}$ of implicit cells of level $i+1$ such that for all $C \in \mathbb{I}$, $C = (\cdot, s \times \cdot, \cdot)$.
Output : A satisfiability-invariant implicit cell C containing s.

$(C_1, \ldots, C_k) :=$ `compute_cover(`$\mathbb{I}$`)` `// [3, Section 4.4.1]`
foreach $j = 1, \ldots, k$ **do**
  $(Q_j, s'_j, \cdot) := C_j$
  compute set Ξ_j of symbolic roots of $\{p \mid sgn\_inv(p) \in Q_j\}$
choose representations $(\mathtt{I}_j, E_j, \preceq)$ of Ξ_j w.r.t. s'_j for $j = 1, \ldots, k$ such that $\preceq$ fulfils the requirement of Definition 8.4
$Q := \cup_{j=1,\ldots,k} Q_j \cup \{cov((\mathtt{I}_1, \ldots, \mathtt{I}_k), s)\}$
apply proof rules to Q considering $\mathtt{I}_j, E_j$ for $j = 1, \ldots, k$ and $\preceq$ until only properties $sgn\_inv(p)$ (of level i) where p is irreducible remain
$C :=$ $(Q, s,$ `compute_cell(`$s, \{p \mid sgn\_inv(p) \in Q\}$`))` `// Algorithm 5`
return C

However, the implementation of our proof system is complete: In case a polynomial is nullified, we add some of its partial derivatives to ensure its order invariance, as suggested in [37, Section 5.2].

9.1.1 Sampling

When assigning a variable in `sample_outside`, we choose the value according to the following scheme: If there are no unsatisfying intervals, we take 0. Otherwise, we chose an integer below all intervals if possible. Otherwise, we choose an integer above all intervals if possible. Otherwise, we choose a sample point between existing intervals; again, we prefer integers or nice rational numbers if possible, as choosing algebraic numbers leads to expensive computations.

9.1.2 Variable Orderings

Variable orderings have a huge impact on the computation of a CAD [17,24,43]. For technical reasons, our implementation supports static variable orderings only, i.e. we determine a fixed variable ordering based on the set of input constraints and do not adapt the ordering during the computation. This ordering determines the order in which the variables are assigned. However, the CAlC method admits to freely choose any variable to be assigned next; exploiting this is part of future work.

For quantifier-free formulas, the static orderings are:

Feature based This class of variable orderings computes a set of features of variables within the set of input polynomials (such as average degree, sum of degrees, ...) and sorts the variables by their features (we sort by one feature, break ties using a second or third feature). The first such heuristic was suggested by Brown [11]. We use a recent improvement obtained using machine-learning techniques from [45].

Max univariate Assuming that all preceding variables have been substituted, we select the variable next in which the most constraints are univariate.

To adapt these orderings for quantified formulas, we apply each ordering separately for each *quantifier block*. Two variables x_i and x_j are in the same quantifier block if and only if $Q_{i'} = Q_{j'}$ for all $i', j' \in \{i, \dots, j\}$ in the prefix $Q_1 x_1 \cdots Q_n x_n$.

9.1.3 Implicants

Experience from [43], where disabling Boolean decisions in our MCSAT implementation performed best, led to the assumption that Boolean reasoning might make unfavorable decisions for non-linear arithmetic problems. Usually, the algebraic part is harder to solve than the Boolean structure (which is often not complex in the corresponding SMT-LIB benchmark set).

To investigate the impact of the "algebraic" complexity on the running time, we implement all three variants for computing implicants (Sect. 6) in a (naive) straight-forward way: We compute all possible implicants and choose the best implicant afterwards (see below).

Evaluation Straight-forward implementation.

Propagation We implement full propagation. In a preprocessing step, we add clauses like $\neg(p < 0) \vee \neg(p > 0)$ to facilitate Boolean propagations: Without this clause, if $p < 0$ would be assumed to be true, although we would conclude that $p \leq 0$ cannot hold by Boolean reasoning, we would not for $p > 0$ causing additional effort in the theory solving.

Exploration Note that this is a rather inefficient implementation of a SAT solver (i.e. without clause-learning etc). Future implementations might consider a more efficient algorithm.

Although we transform the input formula to prenex normal form, we do not eliminate Boolean operators such as exclusive-or and the like, but extend the implicant computation to support these operators.

9.1.4 Inprocessing

Gröbner bases The work in [51] suggest that preconditioning formulas using Gröbner bases speed up CAD computations. We thus use them for inprocessing (Sect. 6.4), which is applied whenever possible.

9.1.5 Implicant Selection Heuristic

After we compute a set of implicants using one of the variants described in Sect. 6, we choose the best according to one of the following criteria:

Size We take the implicant with the minimal number of constraints.

Feature based We take modified features from [45] to choose the best implicant. For every set of constraints, we compute (1) the sum (over all defining polynomials) of the average of the total degrees of the monomials, (2) the average of the total degrees of all monomials., (3) the sum of the total degrees of all monomials. We sort the sets first by (1), breaking ties with (2), breaking ties with (3).

Sum of total degrees (Sotd) We take the implicant with the minimal sum of total degrees, i.e. the sum of total degrees of all monomials of all polynomials in the implicant. The work in [24] suggests that this predicts the size and computation time of a CAD. We break ties using the size of the implicant.

Reverse sotd The opposite of Sum of total degrees (for illustrational purposes).

9.2 Evaluation

We implemented the CAlC algorithm in our SMT solver `SMT-RAT` [21]. The implementation is complete for all discussed problems, i.e. we fully support checking quantifier-free and quantified formulas as well as quantifier elimination. Note that our implementation does not convert the input matrix to conjunctive normal form, but directly works on the Boolean structure. For algebraic computations, we rely on `libpoly` [31]; for factorization and Gröbner bases, we use `CoCoALib` [1]. The default variant of `SMT-RAT` version `24.02` uses CAlC for quantified problems, and our MCSAT implementation for formulas that can be transformed to quantifier-free formulas. For quantifier-free problems, we also apply standard preprocessing techniques. Support for quantifier elimination needs to be enabled using a flag before compiling. The tool is available at https://github.com/ths-rwth/smtrat/.

Our implementation is modular in order to evaluate the described variants. We conduct our experiments on Intel®Xeon®Platinum 8160 CPUs with 2.1GHz per core. We use SMT-LIB's *QF_NRA* (quantifier-free) and *NRA* (with quantifiers) benchmark sets [7]. The source code, instructions for reproducing the experiments and our raw results are available as Supplementary Material.[1]

9.3 Evaluation of Variants

We start by evaluating the variants of our algorithm for checking satisfiability on the *QF_NRA* benchmark set due to the greater amount of (non-trivial) benchmarks. We define a default variant which solved the most instances in preliminary experiments: it uses the Max univariate variable ordering, Propagation for Boolean reasoning, and selects

[1] Also available at https://doi.org/10.5281/zenodo.14355422.

Table 1 Evaluation results of variants

	Boolean reasoning		Selection heuristic			Var. order.	Inproc.	
	Expl.	Eval.	Rev. sotd	Size	Feat. b.	Feat. b.	G.B.	Default
sat	5169	5153	5164	5169	5174	5233	5161	5184
unsat	4547	4738	4985	5025	5046	4588	5047	5048
timeout	2090	1801	1538	1491	1471	1944	1484	1454
memout	328	442	447	449	443	369	442	448
solved	9716	9891	10149	10194	10220	9821	10208	10232

Rows: Number of solved satisfiable and unsatisfiable instances (sat and unsat), their sum (solved), instances where time or memory is exceeded (timeout and memout). The Default column is the variant that uses Propagation for Boolean reasoning, Sotd for the selection heuristic, Max univariate for the variable ordering, and no preprocessing

implicants based on the Sotd criteria. All other variants use this configuration, but vary one of these criteria. The results are shown in Table 1.

The choice of the variable ordering has a high impact. One variable ordering is better on the satisfiable instances while the other is on unsatisfiable ones. Further, Feature Based solves 122 instances not solved by Max Univariate; the other way round, it is 533. Thus, there is potential for future improvements.

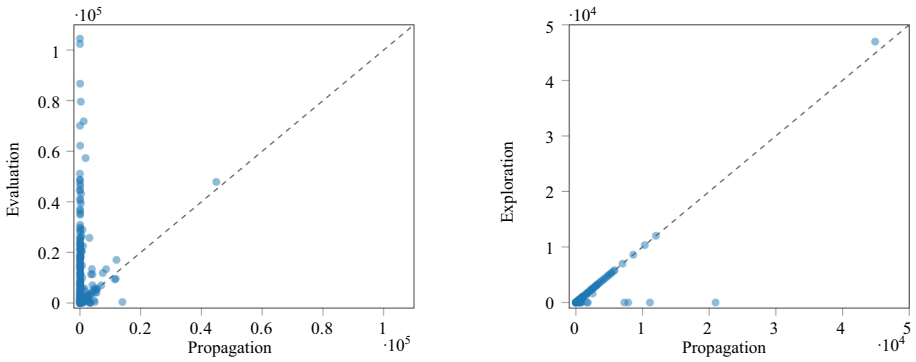
The implicant selection heuristic has a moderate impact, as shown by the numbers of Sotd (see Default column) and its reverse variant Rev. sotd. This indicates a certain variety in the set of implicants from which we select one, but on average only 77% of the generated implicants are used for computing cells. The difference of all meaningful variants (Size, Feature Based, Sotd) is not big, i.e., the virtual best of all variants solves 10250 instances, only 18 more than the Sotd. It is unclear whether significant improvements to this heuristics are possible.

Regarding the Boolean reasoning, we observe that Propagation is better than Evaluation (it solves 354 new instances and loses only 13 instances), however, Exploration performs worse than Evaluation (it gains only 4 instances while losing 520). The first is explained by Fig. 2a: Propagation requires far fewer implicants than Evaluation, likely because conflicts are detected earlier using Boolean propagation. We would expect a similar effect in Fig. 2b, but Exploration very rarely needs fewer implicants than Propagation; instead, profiling reveals that Exploration spends orders of magnitudes more time in the Boolean propagation and exploration than Propagation (Exploration spends 80% of the time for Boolean reasoning on 114 instances (which it solved, all of them in than 5 s); meanwhile, Propagation only spends 1% of the time for Boolean reasoning on 110 of these instances). This is likely due to our rather basic implementation (unoptimized data structures, no clause learning, no watched literals, backtracking is always done to the last UIP instead of the first) to generate *all possible implicants* and choose the best one instead of generating a single good one. Given the large differences on unsat instances in particular, further improvements seem possible.

Also, always applying Gröbner bases to the implicant (as done in the Gröbner variant) results in fewer solved instances than without. As mentioned above, we might need a heuristic that decides when to apply Gröbner bases, as suggested in [28].

9.4 Comparison with Other SMT Solvers

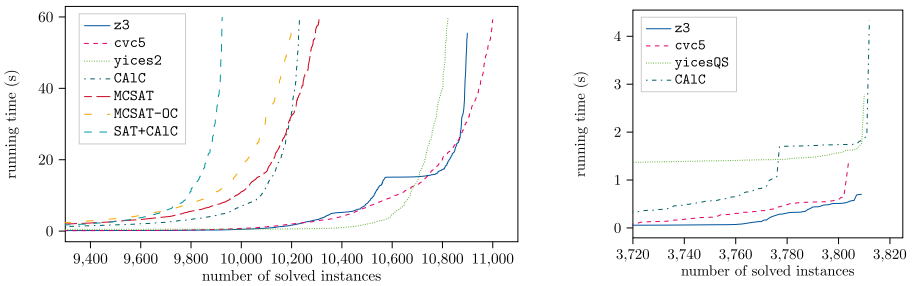
We now compare the best variant (i.e. the default variant) of our algorithm `CAlC` with other solvers, both on *QF_NRA* and *NRA*. For the former, we use `z3` 4.12.4, `cvc5` 1.1.0,



(A) Influence of Boolean propagation.

(B) Influence of Boolean exploration.

Fig. 2 Number of implicants that were used for the covering



(A) Solved instances on the QF_NRA benchmarks.

(B) Solved instances on the NRA benchmarks.

Fig. 3 Performance profiles of SMT solvers

`yices2` 2.6.4, an incremental implementation of the CAlC method in our solver as CDCL(T)-backend `SAT+CAlC` and two MCSAT implementations in our solver, namely `MCSAT-OC` which uses only the single cell construction from [42] for theory solving, and `MCSAT` which additionally uses Fourier-Motzkin, interval constraint propagation [32], virtual substitution [5], and subtropical satisfiability [41]. For the latter, we compare against `z3` 4.12.4, `cvc5` 1.1.0 and `yicesQS` (Oct 22, 2023).

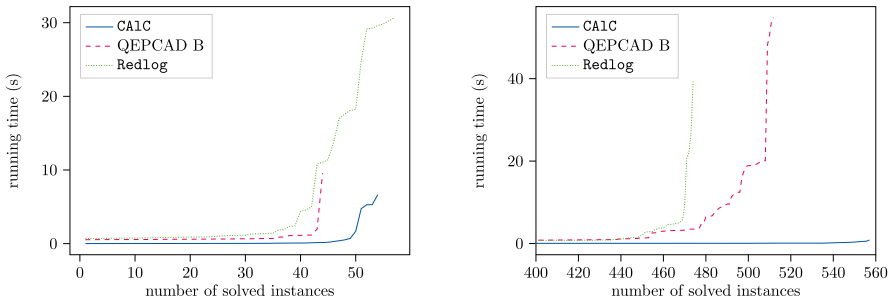
The results for quantifier-free benchmarks are depicted in Fig. 3a. Clearly, all `SMT-RAT` variants solve less than the other solvers, partly due to less efficient data structures in particular for large instances. Further, `cvc5` makes heavy use of linearizations [35], and `yices2` dynamically changes variable orderings during search. `CAlC` is significantly faster than `SAT+CAlC`. `CAlC` solves more instances than `MCSAT-OC` within the given time limit and is generally faster; however, this is expected to change with a higher timeout.

Table 2A compares `MCSAT-OC` and `CAlC`, confirming the impression that `CAlC` is worse on problems with complex Boolean structure, but has a solid advantage on instances containing hard polynomials. Table 2B compares `SAT+CAlC` and `CAlC`, yielding a different picture: The instances solved by both solvers have relatively simple Boolean structures, but the instances solved by only one solver have complex Boolean structures—suggesting that the CAlC method particularly requires "luck" on problems with Boolean structure.

The results for quantified benchmarks in Fig. 3b look very promising for `CAlC`, as it solves 6 instances not solved by `yicesQS` while loosing 4. However, we should not draw

Table 2 Number of instances, their average number of clauses (after converting them to conjunctive normal form), and their average maximum degree of input polynomials, filtered by instances solved by both or only one (and not the other) solver

(A) Comparison with MCSAT			
	Instances solved by		
	MCSAT-OC	Both	CAlC
# instances	248	9958	274
Avg. # clauses	1635	377	320
(B) Comparison with classical CAlC			
	Instances solved by		
	SAT+CAlC	Both	CAlC
# instances	46	9879	353
Avg. # clauses	1975	319	1958
Avg. max. deg	4.5	5.8	3.8



(A) Solved instances on Bath's benchmarks.

(B) Solved instances on the NRA benchmarks from SMT-LIB.

Fig. 4 Performance profile of quantifier elimination tools

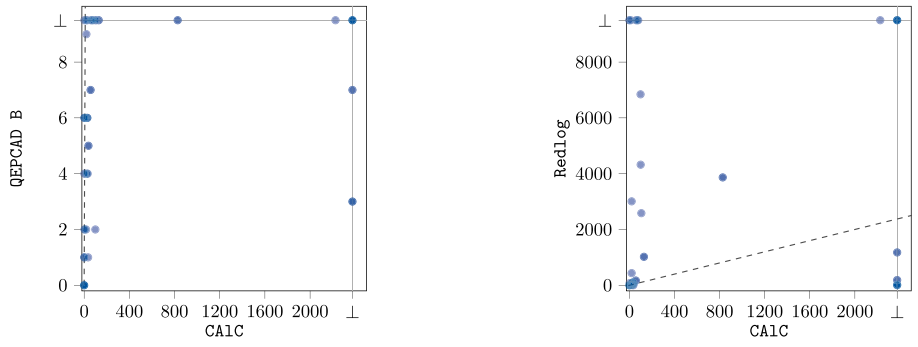
further conclusions because most instances are solved quickly by all solvers and only 3 instances remain unsolved.

We further note that all solvers agreed on the same status for each instance.

9.5 Comparison with Other Quantifier Elimination Tools

Finally, we evaluate `CAlC` against `QEPCAD B` (used as backend through `Tarski` 1.28 [49]) and `Redlog` svn6658. We do not evaluate against commercial tools such as `Maple` or `Mathematica` because of the necessary licences. We evaluate the tools on two benchmark sets: Firstly, we use a collection of CAD examples by David Wilson from Bath University [50], consisting of 78 formulas encoding relevant mathematical statements, including quantifier alternations and parameters. As this set is small, we also use SMT-LIB's *NRA* benchmarks; some of these instances contain rational functions and other peculiar features of SMT-LIB, and we only use those instances which can be converted to inputs for `QEPCAD B` and `Redlog` in a straight-forward way. All scripts for converting these benchmarks to the respective input formats are provided in the Zenodo repository.

The running times are depicted in Fig. 4a and b. We note that `QEPCAD B` fails on 25 instances due to the incompleteness of McCallum's projection. `CAlC` is competitive on the Bath benchmarks, solving 9 benchmarks more than `QEPCAD B` and 4 benchmarks less than `Redlog`. On the SMT-LIB benchmarks, `CAlC` outperforms the other solvers significantly.



(A) CAlC vs QEPCAD B. (B) CAlC vs Redlog.

Fig. 5 Quality of the solution formula: we compare the number of atoms in the solution formula (indicated by the coordinates). Every point that is not on one of the gray lines represents an instance solved by both solvers. The ⊥ line indicates that the corresponding solver did time out on the instance

To measure the quality of the generated solution formulas describing the parameter space, we count the number of atoms, as depicted in Fig. 5a and b. Clearly, QEPCAD B is superior to CAlC. Still, CAlC produces significantly smaller solution formulas than Redlog on many instances.

To verify the correctness of the solution formulas produced by CAlC, we used Tarski 1.40 to check whether they are equivalent to the input problems. To our knowledge, Tarski is the only system that supports indexed root expressions in the input, which may be contained in CAlC's solution formulas.

10 Conclusion and Future Work

10.1 Future Work

Preprocessing The preprocessing techniques implemented in SMT-RAT are designed for quantifier-free formulas. Incorporating incomplete techniques such as *virtual substitution* for quantified formulas might help to improve the performance on some benchmarks.

Dynamic Variable Orderings Variable orderings have a crucial impact on the performance of CAD-based algorithms. The CAlC algorithm allows for different variable orderings for every branch and the algorithm can naturally combine sub-results. This is technically possible in MCSAT as well—however, combining sub-results computed with different variable orderings comes with high costs when combining projection results that stem from different variable orderings. The advantages of CAlC could be further facilitated by employing the techniques described in Sect. 7 to split the input into more branches, where, again, a different variable ordering is possible in each branch.

Implicant Calculation The experimental evaluation shows that the computation of implicants plays a central role in the algorithm. While the choice based on algebraic criteria (e.g. degrees of polynomials) is important, (efficient) Boolean reasoning is crucial. Our implementation features only a basic implementation for Boolean reasoning which exhaustively computed all possible conflicts in order to choose the "best" one according to algebraic criteria. Future implementations should focus on efficient Boolean reasoning, incorporating techniques from SAT solvers—or even using a SAT solver—guided by algebraic criteria in order to compute a single implicant which is "good".

Reduce Lifting over Sections The work in [6] extends the CAlC algorithm for closed cells, i.e., it tracks whether a truth-invariant cell maintains the same truth value on its closure, based on strict relation symbols in the input formula. This allows to build coverings with closed intervals in the CAlC algorithm. Thereby, we could avoid exploring the branches on some cell boundaries which oftentimes involve computations with non-rational real algebraic numbers that are particularly computationally expensive. This technique could be extended to CAlC for quantifiers in a straight-forward way.

Minimizing Solution Formulas for Quantifier Elimination The CAlC algorithm has a competitive running time for quantifier elimination, however, in particular `QEPCAD B` computes significantly smaller solution formulas for the parameters. With some effort, the techniques employed by `QEPCAD B` to minimize the solution formula could also be applied for results computed by CAlC.

Optimization Problems *Optimization Modulo Theories* [8] deals with the optimization variant of SMT, where we are not only interested in *some* solution that satisfies the input formula, but a solution where the value of a specified *objective variable* is minimal or maximal. A naive approach would be to transform the problem to a quantifier elimination problem where the objective variable is the only parameter. We then compute its solution space and pick the minimal or maximal value. Less naively, we could compute only the part of its solution space that is sufficient to prove that a certain value is the minimal or maximal value.

10.2 Conclusion

We generalized the successful CAlC algorithm to quantified input formulas and quantifier elimination problems. Our algorithm works directly on formulas with arbitrary Boolean and quantifier structure by shifting Boolean reasoning to computation of implicants that explain a conflict. This avoids the need for a complex CDCL(T) architecture and keeps the implementation relatively simple and compact.

Further, we discussed various extensions of our basic algorithm in order to improve the running times and to reduce the solution formula size for quantifier elimination. We presented an embedding into a proof system for cylindrical algebraic decomposition which allows generating certificates to verify the algorithm's results in future implementations.

Our algorithm uses an adapted concept of implicants to incorporate Boolean reasoning. Their computation plays a central role for the performance of the algorithm. For our implementation, we focused on generating optimal implicants with respect to algebraic criteria, mostly neglecting the efficiency of Boolean reasoning. We investigated different ways for computing implicants, varying in the exhaustiveness of Boolean reasoning and the algebraic criteria. Although it was shown experimentally that algebraic criteria have an impact, the computational effort spent on Boolean reasoning limits the scalability of the current implementation.

Still, our algorithm shows decent performance compared to other tools: On SMT-LIB's QF_NRA benchmarks, it is comparable with our MCSAT implementation, while state-of-the-art SMT solvers outcompete both implementations. On SMT-LIB's NRA benchmarks, our algorithm solves the most instances, outcompeting the state-of-the-art SMT solvers. Compared to quantifier elimination tools, our algorithm is competitive with

respect to running times. However, regarding the output formula size, the state-of-the-art tool `QEPCAD B` produces smaller formulas.

Particularly the latter comparison shows that we successfully transferred ideas from SMT solving to quantifier elimination. The results are promising and motivate future work on the algorithm. Throughout the paper, we presented various possibilities for further improving its performance.

Acknowledgements
Thanks to Erika Ábrahám for discussions on the CAIC method, and to Christopher W. Brown for the discussion on indexed root expressions and Thom's lemma, as well as extending Tarski to support indexed root expression in the input formulas. We thank Philip Kroll for initiating the work on Sect. 7 and implementing parts of this work in his thesis.

Author Contributions
All authors contributed to all sections of the paper.

Funding
Open Access funding enabled and organized by Projekt DEAL. 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*.

Data Availability
Data is provided as Supplementary Material with the online version of this article.

Declarations

Conflict of Interest
The authors declare that they have no conflict of interest.

Received: 23 February 2024 Revised: 3 September 2024 Accepted: 11 November 2024
Published online: 28 April 2026

References

1. Abbott, J., Bigatti, A.M.: CoCoALib: a C++ library for doing computations in commutative algebra. https://cocoa.dima.unige.it/cocoa/cocoalib
2. Abraham, E., Davenport, J.H., England, M., Kremer, G.: Proving UNSAT in SMT: the case of quantifier free non-linear real arithmetic (2021). https://doi.org/10.48550/arXiv.2108.05320
3. Ábrahám, E., Davenport, J.H., England, M., Kremer, G.: Deciding the consistency of non-linear real arithmetic constraints with a conflict driven search using cylindrical algebraic coverings. J. Logic. Algebraic Methods Program. (2021). https://doi.org/10.1016/j.jlamp.2020.100633
4. Ábrahám, E., Davenport, J.H., England, M., Kremer, G., Tonks, Z.: New opportunities for the formal proof of computational real geometry? In: Practical Aspects of Automated Reasoning and Satisfiability Checking and Symbolic Computation (PAAR+SC-Square 2020), CEUR Workshop Proceedings (2020). http://ceur-ws.org/Vol-2752/paper13.pdf
5. Ábrahám, E., Nalbach, J., Kremer, G.: Embedding the virtual substitution method in the model constructing satisfiability calculus framework (work-in-progress paper). In: Satisfiability Checking and Symbolic Computation (SC-square 2017), CEUR Workshop Proceedings (2017). https://ceur-ws.org/Vol-1974/EAb.pdf
6. Bär, P., Nalbach, J., Ábrahám, E., Brown, C.W.: Exploiting strict constraints in the cylindrical algebraic covering. In: Satisfiability Modulo Theories (SMT 2023), CEUR Workshop Proceedings (2023). https://ceur-ws.org/Vol-3429/paper13.pdf
7. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB standard: version 2.6. Technical report, Department of Computer Science, The University of Iowa (2017)
8. Bigarella, F., Cimatti, A., Griggio, A., Irfan, A., Jonáš, M., Roveri, M., Sebastiani, R., Trentin, P.: Optimization modulo non-linear arithmetic via incremental linearization. In: Konev, B., Reger, G. (eds.) Frontiers of Combining Systems, vol. 12941. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-86205-3_12
9. Bjørner, N., Janota, M.: Playing with quantified satisfaction. In: Logic for Programming, Artificial Intelligence and Reasoning (LPAR-20) (2015). https://doi.org/10.29007/vv21
10. Bonacina, M.P., Graham-Lengrand, S., Vauthier, C.: QSMA: a new algorithm for quantified satisfiability modulo theory and assignment. In: Automated Deduction (CADE 29). Lecture Notes in Computer Science (2023). https://doi.org/10.1007/978-3-031-38499-8_5
11. Brown, C.W.: Companion to the tutorial "Cylindrical algebraic decomposition'. Presented at ISSAC '04, (2004). https://www.usna.edu/Users/cs/wcbrown/research/ISSAC04/handout.pdf
12. Brown, C.W.: Guaranteed solution formula construction. In: International Symposium on Symbolic and Algebraic Computation (ISSAC 1999) (1999). https://doi.org/10.1145/309831.309890
13. Brown, C.W.: Improved projection for cylindrical algebraic decomposition. J. Symb. Comput. (2001). https://doi.org/10.1006/jsco.2001.0463
14. Brown, C.W.: Open Non-uniform Cylindrical Algebraic Decompositions. In: International Symposium on Symbolic and Algebraic Computation (ISSAC '15) (2015). https://doi.org/10.1145/2755996.2756654

15. Brown, C.W.: QEPCAD B: a program for computing with semi-algebraic sets using CADs. ACM SIGSAM Bull. (2003). https://doi.org/10.1145/968708.968710
16. Brown, C.W.: Solution formula construction for truth invariant CAD's. PhD thesis, University of Delaware (1999). https://doi.org/10.5555/929495
17. Brown, C.W., Davenport, J.H.: The complexity of quantifier elimination and cylindrical algebraic decomposition. In: International Symposium on Symbolic and Algebraic Computation (ISSAC'07) (2007). https://doi.org/10.1145/1277548.1277557
18. Brown, C.W., Košta, M.: Constructing a single cell in cylindrical algebraic decomposition. J. Symb. Comput. (2015). https://doi.org/10.1016/j.jsc.2014.09.024
19. Brown, C.W., Vale-Enriquez, F.: From simplification to a partial theory solver for non-linear real polynomial constraints. J. Symb. Comput. (2020). https://doi.org/10.1016/j.jsc.2019.07.020
20. Collins, G.E.: Quantifier elimination for real closed fields by cylindrical algebraic decomposition. Automata Theory and Formal Languages (1975). https://doi.org/10.1007/3-540-07407-4_17
21. Corzilius, F., Kremer, G., Junges, S., Schupp, S., Ábrahám, E.: SMT-RAT: an open source C++ toolbox for strategic and parallel SMT solving. In: Theory and applications of satisfiability testing (SAT 2015). Lecture Notes in Computer Science (2015). https://doi.org/10.1007/978-3-319-24318-4_26
22. de Moura, L., Bjørner, N.: Efficient e-matching for SMT solvers. In: Automated Deduction (CADE 21). Lecture Notes in Computer Science (2007). https://doi.org/10.1007/978-3-540-73595-3_13
23. de Moura, L., Jovanović, D.: A model-constructing satisfiability calculus. In: Verification, Model Checking, and Abstract Interpretation (VMCAI 2013). Lecture Notes in Computer Science (2013). https://doi.org/10.1007/978-3-642-35873-9_1
24. Dolzmann, A., Seidl, A., Sturm, T.: Efficient projection orders for CAD. In: International Symposium on Symbolic and Algebraic Computation (ISSAC'04) (2004). https://doi.org/10.1145/1005285.1005303
25. England, M., Davenport, J.H.: The Complexity of cylindrical algebraic decomposition with respect to polynomial degree. In: Computer Algebra in Scientific Computing (CASC 2016). Lecture Notes in Computer Science (2016). https://doi.org/10.1007/978-3-319-45641-6_12
26. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): fast decision procedures. In: Computer Aided Verification (2004). https://doi.org/10.1007/978-3-540-27813-9_14
27. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiability modulo theories. In: Computer Aided Verification (CAV 2009). Lecture Notes in Computer Science (2009). https://doi.org/10.1007/978-3-642-02658-4_25
28. Huang, Z., England, M., Davenport, J.H., Paulson, L.C.: Using machine learning to decide when to precondition cylindrical algebraic decomposition with Groebner bases. In: Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2016) (2016). https://doi.org/10.1109/SYNASC.2016.020
29. Iwane, H., Yanami, H., Anai, H., Yokoyama, K.: An effective implementation of a symbolic-numeric cylindrical algebraic decomposition for quantifier elimination. In: Proceedings of the 2009 Conference on Symbolic Numeric Computation, SNC '09 (2009). https://doi.org/10.1145/1577190.1577203
30. Jovanović, D., de Moura, L.: Solving non-linear arithmetic. In: Automated Reasoning (IJCAR 2012). Lecture Notes in Computer Science (2012). https://doi.org/10.1007/978-3-642-31365-3_27
31. Jovanović, D., Dutertre, B.: LibPoly: A Library for Reasoning about Polynomials. In: Satisfiability Modulo Theories (SMT 2017), CEUR Workshop Proceedings (2017). https://ceur-ws.org/Vol-1889/paper3.pdf
32. Kremer, G.: Cylindrical algebraic decomposition for nonlinear arithmetic problems. PhD thesis, RWTH Aachen University (2020). https://doi.org/10.18154/RWTH-2020-05913
33. Kremer, G., Ábrahám, E., England, M., Davenport, J.H.: On the implementation of cylindrical algebraic coverings for satisfiability modulo theories solving. In: Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2021) (2021). https://doi.org/10.1109/SYNASC54541.2021.00018
34. Kremer, G., Nalbach, J.: Cylindrical algebraic coverings for quantifiers. In: Satisfiability Checking and Symbolic Computation 2022 (SC-square 2022), CEUR Workshop Proceedings (2023). https://ceur-ws.org/Vol-3458/paper1.pdf
35. Kremer, G., Reynolds, A., Barrett, C., Tinelli, C.: Cooperating techniques for solving nonlinear real arithmetic in the cvc5 SMT solver (system description). In: Automated Reasoning (IJCAR 2022). Lecture Notes in Computer Science (2022). https://doi.org/10.1007/978-3-031-10769-6_7
36. Lazard,D.: An improved projection for cylindrical algebraic decomposition. In Bajaj, C.L. (ed.) Algebraic Geometry and Its Applications: Collections of Papers from Shreeram S. Abhyankar's 60th Birthday Conference. Springer, New York, NY (1994). https://doi.org/10.1007/978-1-4612-2628-4_29
37. McCallum, S.: An improved projection operation for cylindrical algebraic decomposition. PhD thesis, University of Wisconsin-Madison (1985)
38. McCallum, S.: An improved projection operation for cylindrical algebraic decomposition. In: Quantifier Elimination and Cylindrical Algebraic Decomposition (1998). https://doi.org/10.1007/978-3-7091-9459-1_12
39. McCallum, S., Parusiński, A., Paunescu, L.: Validity proof of Lazard's method for CAD construction. J. Symb. Comput. (2019). https://doi.org/10.1016/j.jsc.2017.12.002
40. Mishra, B.: Algorithmic Algebra. Springer, New York (1993). https://doi.org/10.1007/978-1-4612-4344-1
41. Nalbach, J., Ábrahám, E.: Subtropical satisfiability for SMT solving. In: NASA Formal Methods (NFM 2023). Lecture Notes in Computer Science (2023) https://doi.org/10.1007/978-3-031-33170-1_26
42. Nalbach, J., Ábrahám, E., Specht, P., Brown, C.W., Davenport, J.H., England, M.: Levelwise construction of a single cylindrical algebraic cell. J. Symb. Comput. (2024). https://doi.org/10.1016/j.jsc.2023.102288
43. Nalbach, J., Kremer, G., Ábrahám, E.: On variable orderings in MCSAT for non-linear real arithmetic. In: Satisfiability Checking and Symbolic Computation (2019) (SC-square 2019), CEUR Workshop Proceedings 2019. https://ceur-ws.org/Vol-2460/paper5.pdf
44. Niemetz, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Syntax-guided quantifier instantiation. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2021). Lecture Notes in Computer Science (2021). https://doi.org/10.1007/978-3-030-72013-1_8

45. Pickering, L., del Río Almajano, T., England, M., Cohen, K.: Explainable AI insights for symbolic computation: a case study on selecting the variable ordering for cylindrical algebraic decomposition. J. Symb. Comput. (2024). https://doi.org/10.1016/j.jsc.2023.102276
46. Seidl, A., Sturm, T.: A generic projection operator for partial cylindrical algebraic decomposition. In: International Symposium on Symbolic and Algebraic Computation (ISSAC '03) (2003). https://doi.org/10.1145/860854.860903
47. Strzeboński, A.: Solving systems of strict polynomial inequalities. J. Symb. Comput. (2000). https://doi.org/10.1006/jsco.1999.0327
48. Tarski, A.: A decision method for elementary algebra and geometry (1951). https://doi.org/10.1007/978-3-7091-9459-1_3
49. Vale-Enriquez, F., Brown, C.W.: Polynomial constraints and unsat cores in Tarski. In: Davenport, J.H., Kauers, M., Labahn, G., Urban, J. (eds.) Mathematical Software—ICMS 2018, Vol. 10931. Springer, Cham, (2018). https://doi.org/10.1007/978-3-319-96418-8_55
50. Wilson, D.: Real geometry and connectedness via triangular description: CAD example bank (2013). https://researchdata.bath.ac.uk/69/
51. Wilson, D.J., Bradford, R.J., Davenport, J.H.: Speeding up cylindrical algebraic decomposition by Gröbner bases. In: Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Jeuring, J., Campbell, J.A., Carette, J., Dos Reis, G., Sojka, P., Wenzel, M., Sorge, V. (eds.) Intelligent Computer Mathematics, vol. 7362. Springer, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31374-5_19

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.