Fully Incremental Cylindrical Algebraic Decomposition

Gereon Kremer

Theory of Hybrid Systems, RWTH Aachen University, 52056 Aachen, Germany

Erika Ábrahám

Theory of Hybrid Systems, RWTH Aachen University, 52056 Aachen, Germany

Abstract

Collins introduced the *cylindrical algebraic decomposition* method for eliminating quantifiers in real arithmetic formulas. In our work we use this method for satisfiability checking in *satisfiability modulo theories* solver technologies, and tune it by trying to avoid some computation steps that are needed for quantifier elimination but not for satisfiability checking. We further propose novel data structures and adapt the method to work *incrementally* and to support *backtracking*. We verify the effectiveness experimentally by comparing different versions of the cylindrical algebraic decomposition used within a state-of-the-art satisfiability modulo theories solver.

Keywords: cylindrical algebraic decomposition, incrementality, backtracking, satisfiability modulo theories

1. Introduction

The concept of *quantifier elimination* is a powerful tool for dealing with (quantified) formulas which aims to construct an equivalent quantifier-free formula for a given formula with quantified variables. The *cylindrical algebraic decomposition* method was proposed by Collins (1975) as a generic quantifier elimination procedure for the *elementary theory of real closed fields*. We call this theory, which is concerned with equalities and inequalities of polynomials over real variables, *nonlinear real arithmetic*. Since its introduction, numerous works on the theory, implementation and applications of the cylindrical algebraic decomposition method led to elegant improvements and useful extensions.

In this paper we do not consider the general case but focus on fully existentially quantified problems where all variables are (explicitly or implicitly) quantified existentially. The motivation for this work stems from the application of the cylindrical algebraic decomposition method to *satisfiability checking*, more specifically *satisfiability modulo theories (SMT) solving* as presented by Barrett et al. (2009). In the case of satisfiability, this application requires only a single

Email addresses: gereon.kremer@cs.rwth-aachen.de (Gereon Kremer), abraham@cs.rwth-aachen.de (Erika Ábrahám)

URL: https://ths.rwth-aachen.de/people/gereon-kremer/ (Gereon Kremer), https://ths.rwth-aachen.de/people/erika-abraham/ (Erika Ábrahám)

satisfying solution to prove satisfiability, therefore a procedure tuned for this purpose can provide a great benefit to SMT solvers.

Due to its inherent mathematical complexity, the use of cylindrical algebraic decompositions is not particularly wide-spread in this community, but it has already proven to be very useful. A rather direct integration into an SMT solver was presented by Corzilius et al. (2015) which mentions an earlier version of the work presented here. It was subsequently extended to the level of incrementality presented here, but also to allow the solving of integer problems by Kremer et al. (2016), and combined with other solving methods, for example with *interval constraint propagation* by Loup et al. (2013). Another significant contribution is the very successful *NLSAT* approach by Jovanović and De Moura (2012) that includes a novel adaption of the cylindrical algebraic decomposition method in the context of satisfiability checking.

To make the cylindrical algebraic decomposition method competitive in the SMT solving context, it should be adapted to fulfill some specific requirements. Given a set of polynomial constraints, it either has to find a single satisfying solution or prove that there is no solution. In this scenario the computation can be stopped when a solution is found, therefore we want to adapt our perception of the cylindrical algebraic decomposition method to regard it a *search method* for satisfying solutions. Furthermore we usually do not solve a single input problem, instead in the SMT context we rather solve a sequence of *related* problems. Retaining already computed information and reusing it for the next problem can be essential to effectively solve real problems. We aim at satisfying these requirements by a variant of the cylindrical algebraic decomposition method that can work *incrementally* and may *backtrack*, two concepts that we explain in more detail in Section 3.

Our approach essentially reorders the computations of the cylindrical algebraic decomposition method and allows for interrupting and continuing these computations. The different projection operators or a heuristic variable ordering can all be used within the proposed adaption. Thus, issues like correctness or termination immediately carry over and we refrain from looking at the mathematical foundations in this paper.

We first introduce the notations used in the paper and give a high-level introduction to the cylindrical algebraic decomposition method in Section 2. Section 3 then defines the framework in which this method is used and gives some more details on our motivation and the requirements on our methods. Subsequently we propose our adapted version of the cylindrical algebraic decomposition internals starting with the lifting phase in Section 4 and the data structure for the projection phase in Section 5. The algorithms for the projection phase are presented in Section 6 and Section 7. Based on this we specify how backtracking is performed in Section 8 to finally get a complete version of the SMT-adaptation of the cylindrical algebraic decomposition method in Section 9. In Section 10 we discuss some further topics of interest, including techniques to ensure soundness and optimizations that were not explicitly considered in the previous sections. We finally provide experimental evaluations in Section 11 and conclude the paper in Section 12.

2. Preliminaries

2.1. Notations

We start with introducing some notions and notations used throughout the paper. We denote the set of *integers* by $\mathbb{Z}$, the set of *real algebraic numbers* by $\mathcal{R}$ and the *power set* of some set S by $\mathcal{P}(S)$. We assume a set $X = \{x_1, \ldots, x_n\}$ of *variables* with a fixed *total ordering* $x_1 < \ldots < x_n$, and define $X_i = \{x_1, \ldots, x_i\}$ for $1 \leq i \leq n$.

We write $K[y_1, \ldots, y_m]$ for the set of polynomials over variables $\{y_1, \ldots, y_m\}$ with coefficients from K and define $\mathbb{P} = \mathbb{Z}[x_1, \ldots, x_n]$ and $\mathbb{P}_i = \mathbb{Z}[x_1, \ldots, x_i]$ for $1 \leq i \leq n$. We define $P_i = P \cap (\mathbb{P}_i \setminus \mathbb{P}_{i-1})$ for a set of polynomials $P \subseteq \mathbb{P}$. An *input formula* is of the form $\varphi = p_1 \sim_1 0 \wedge \ldots \wedge p_k \sim_k 0$ with polynomials $p_i \in \mathbb{P}$ and $\sim_i \in \{<, \leq, =, \neq, \geq, >\}$ for some $k \in \mathbb{Z}$, $k \geq 1$. We call $C = \{p_1 \sim_1 0, \ldots, p_k \sim_k 0\}$ the *input constraints* and $P = \{p_1, \ldots, p_k\}$ the *input polynomials*.

A *variable assignment* over some $\mathcal{X}' \subseteq \mathcal{X}$ is a function $\alpha : \mathcal{X}' \to \mathcal{R}$. For $p \in \mathbb{P}_i$ we can *evaluate* p under an assignment $\alpha : \mathcal{X}_i \to \mathcal{R}$ to a real algebraic number and likewise evaluate the constraint $p \sim 0$ to either true or false. Evaluating a polynomial $q \in P_{i+1}$ under $\alpha : \mathcal{X}_i \to \mathcal{R}$ yields a univariate polynomial from $\mathcal{R}[x_{i+1}]$ whose real roots are called the *roots of q under α*.

2.2. The Cylindrical Algebraic Decomposition Method

Originally the *cylindrical algebraic decomposition* (*CAD*) method was proposed to work on sets of polynomials and mostly relies on constructing regions that are sign-invariant with respect to these polynomials. It was quickly realized that this implies thruth-invariance with respect to formulas involving these polynomials, giving rise to what we now call *truth-table invariant CAD* Bradford et al. (2016) (*TTICAD*) and most of what we present here is based on this notion of TTICAD. For a given input formula, CAD is able to iteratively eliminate variables from the formula and answer questions concerning the formula's satisfiability or validity. We defined input formulas to be conjunctive because our SMT application uses CAD to determine the satisfiability of *sets* (or conjunctions) of constraints only. Note that thus checking the satisfiability of conjunctions of constraints is sufficient to check whether a fully existentially quantified sentence is true using the SMT framework.

Before we present our SMT adaptation of the CAD method, we first give a rough sketch of how the CAD method works. Note that we do not attempt to thoroughly explain the mathematical background and refer to Collins (1975) or Collins and Hong (1991) for more details.

The fundamental idea of the CAD method is to decompose the space of possible values of the real-valued variables into a finite number of *connected regions* such that *all points* within a region are *equivalent*, where different CAD versions might give different meanings to this equivalence relation. The regular *sign-invariant* CAD considers two points equivalent if all input polynomials evaluate to the same sign under both points. The TTICAD Bradford et al. (2016) already mentioned above considers two points equivalent if the evaluation of the formulae is the same. The TTICAD method computes such a *finite* decomposition represented by a single *sample point* from every region, which allows us to determine satisfiability or validity of the formula by only considering a finite number of points. In the following we use the term CAD for the algorithm, the set of regions or the set of points representing these regions interchangeably.

The core of the CAD method works on the polynomials of the input formula only and ignores the constraints and the formula structure, except for the evaluation in Algorithm 4. The abstract procedure is visualized in Figure 1 and roughly decomposes into the *projection* and the *lifting* phases on the left and right-hand side, respectively. Note that also other terminology is used in the literature, for example *elimination* instead of projection or *construction* instead of lifting. For a CAD of *dimension n* we expect a finite set $P \subseteq \mathbb{P}_n$ of input polynomials and associate the variable x_i with every *level* $1 \leq i \leq n$, a finite set $P_i \subset \mathbb{Z}[x_1, \ldots, x_i]$ of polynomials and a finite set $Z_i \subset \mathbb{R}^i$ of sample points for $(x_1, \ldots, x_i)$. The levels are *decreasing* during the projection and *increasing* during the lifting. We call sample points of dimension *n full-dimensional* and sample points of lower dimensions *partial*.

The projection starts with the set P_n of input polynomials over n variables and uses a *projection operator* to eliminate one variable at a time until it obtains P_1 containing only univariate

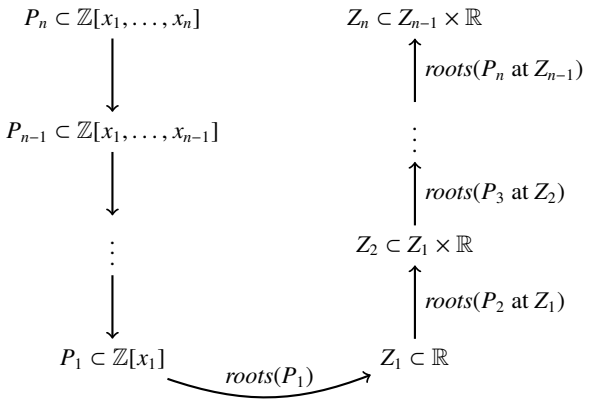


Figure 1: The structure of the CAD method

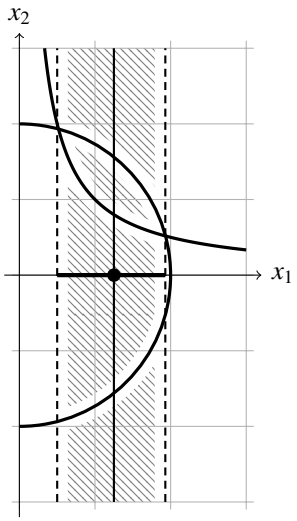


Figure 2: A cylinder in a CAD

polynomials. Once this is done, a set Z_1 of one-dimensional partial sample points for x_1 is constructed. To do this we obtain the set of all real roots of all polynomials in P_1 and extend it by some value smaller than the smallest root, a value between every two successive roots and a value greater than the largest root. Note that these sample points represent each sign-invariant region for P_1. Now every partial sample point is *lifted* by substituting it into all polynomials from P_2 – which yields a set of univariate polynomials in x_2 – and repeating the same process as we have done for the univariate partial sample points to obtain Z_2. This process is iterated until we obtain sample points of full dimension n.

The varieties (sets of real zeros) of the polynomials in $P_1, \ldots, P_n$ decompose the whole $\mathbb{R}^n$ into a finite number of connected regions of equivalent points, where the properties of the projection operator induce the notion of equivalence. The lifting process constructs regions that are arranged in *cylinders* which means that the $(k-1)$-dimensional projections of every two regions from dimension k are either identical or disjoint. The projection operators used for the CAD method (should) assure that in the lifting phase the choice of a point between two zeros does not influence which regions can be covered by the resulting sample and thereby the soundness of the overall approach.

Consider the 2-dimensional example in Figure 2 which shows a cylinder from a CAD partitioned into four shaded regions that are separated by three segments of the polynomial varieties. The projection of all of these seven regions onto the x_1 axis is identical and this interval is indicated by the bold line on the x_1 axis. No matter which partial sample point we select from this interval, it can be extended to sample points from all seven regions.

We observe that the *boundaries* of the cylinders are given by the roots of the lower-dimensional polynomials from P_1. Hence we have to ensure that these polynomials have roots at all relevant places such that we get such a cylindrical alignment of all regions. Different *projection operators* are known that satisfy this property. We give the definition of one of the most popular such operators due to McCallum (1998).

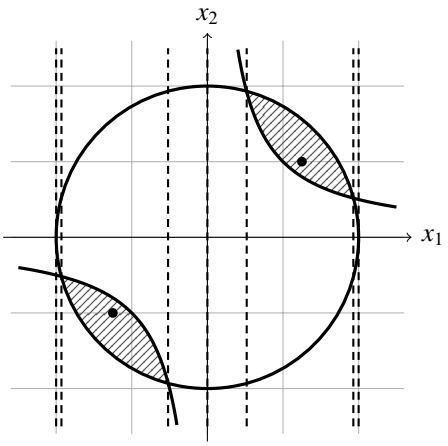


Figure 3: CAD for $x_1^2 + x_2^2 < 4 \wedge x_1 \cdot x_2 > 1$

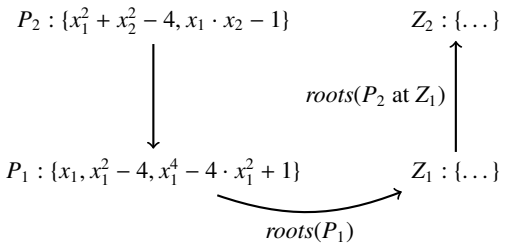


Figure 4: Example computation for $x_1^2 + x_2^2 - 4$ and $x_1 \cdot x_2 - 1$

Definition 1 (McCallum's projection operator)**.** *Given a set of input polynomials P, the* projection set $Proj_x(P)$ due to McCallum *is defined as*

$$Proj_x(P) = \left(\cup_{p \in P} \text{coeffs}_x(p)\right) \cup \{\text{disc}_x(p) \mid p \in P\} \cup \{\text{res}_x(p, q) \mid p, q \in P\}$$

where coeffs_x, disc_x *and* res_x *return the set of all coefficients, the discriminant and the resultant, respectively, with respect to some fixed variable x.*

Note that some projection operators (including the one defined above) only provide soundness under some additional conditions. In particular P may not be any set of polynomials but should rather be a *square-free basis* of some polynomials combined with their contents. This is necessary not only for the input polynomials, but at every level the respective polynomials need to form such a square-free basis. We achieve this goal by using a full factorization of every individual polynomial instead of the (unfactorized) polynomial, both for input polynomials and the results of coeffs_x, disc_x and res_x. Also we may be forced to construct additional *delineating polynomials* if certain cases occur during the lifting process. We do not discuss these issues here further and refer to McCallum (1998); Viehmann et al. (2017) for more details, as well as to Sections 10.5 and 10.6 for some discussion on how to deal with them. We also want to refer to Lazard (1994); Brown (2001) for other projection operators that produce even smaller sets of projection polynomials in practice.

A comparison with several other projection operators can be found in Viehmann et al. (2017). As we do not want to deal with this question here we use a slightly modified version of the projection operator due to McCallum (1998) for all examples and experiments. Our modification is concerned with the set $\text{coeffs}_x(p)$ and checks whether each of the coefficients may vanish, i.e., become zero. If we can show that some coefficient c from a term $c \cdot x^k$ does not vanish we remove all coefficients for smaller k from $\text{coeffs}_x(p)$.

To illustrate this process we consider the input polynomials $P_2 = \{x_1^2 + x_2^2 - 4,\ x_1 \cdot x_2 - 1\}$, an example taken from Brown (2001). Figure 3 shows the varieties of the two polynomials as solid lines and indicates the borders of the one-dimensional cylinders by dashed lines. There are two regions (marked by stripes) that satisfy the two constraints that give rise to the polynomials. The

two sample points that our solver SMT-RAT – presented by Corzilius et al. (2015) – generates are shown as dots.

Figure 4 shows the computations performed for the example from Figure 3. Starting with the two polynomials in P_2 we apply the projection operator to obtain three univariate polynomials in P_1. The real roots of the polynomials in P_1 are $\{-2, \xi_1, \xi_2, 0, \xi_3, \xi_4, 2\}$ (in ascending numerical order) where ξ_i denote the roots of $x_1^4 - 4 \cdot x_1^2 + 1$. Note that each of these roots corresponds to one of the dashed lines. The set of roots is extended by intermediate sample points that represent the cylinders between the dashed lines to obtain Z_1, for example

$$Z_1 = \{-3, -2, -1.95, \xi_1, -1, \xi_2, -0.25, 0, 0.25, \xi_3, 1, \xi_4, 1.95, 2\} .$$

Each of these points from Z_1 is then substituted into the two original polynomials. When using the sample point 1 for example, we obtain the polynomials $x_2^2 - 3$ and $x_2 - 1$ and their real roots $\{-\sqrt{3}, 1, \sqrt{3}\}$. We construct Z_2 by extending the sample point 1 by these roots and the intermediate sample points, for example

$$\{(1, -2), (1, -\sqrt{3}), (1, 0), (1, 1), (1, 1.5), (1, \sqrt{3}), (1, 2)\} .$$

Continuing this process for all samples from Z_1 yields 83 sample points overall in Z_2 (which we do not list here). Note that the roots ξ_i as well as $-\sqrt{3}$ and $\sqrt{3}$, are not rational numbers and we need some exact representation for them. More precisely, we need exact (symbolic) representations for *real algebraic numbers*, which are real roots of univariate polynomials. We refrain from discussing this issue here and refer to Collins and Loos (1983) or Coste and Roy (1988) and only note that our implementation uses isolating intervals based on Descartes' rule of signs.

Most techniques presented in this paper naturally generalize to existentially quantified formulas in negation normal form and the restriction to sets of constraints only simplifies the presentation. The only exception is the evaluation of sample points that we borrow from Collins and Hong (1991) in Algorithm 4 that implicitly assumes the input to be a set of constraints. To generalize this we essentially need to give up on caching the evaluation results for each constraint – though we could still cache evaluation results for subformulas.

3. An SMT-Compliant CAD Adaptation

Before giving details on the data structures and algorithms for our CAD adaptation, let us state some relevant aspects of the context in which our implementation is meant to be used. First and foremost, we do not aim at a general-purpose CAD but a specialization that can be understood as a CAD-based search procedure for satisfying solutions of formulas. Our CAD adaptation attempts to find a single satisfying sample point as fast as possible, and if it finds one the search can be stopped. Furthermore, it allows for efficient solving of a sequence of similar formulas. It does not attempt to improve the case of general quantifier elimination, though it can also be used for that purpose as described in Section 10.8.

3.1. CAD Interface

This work is mainly motivated by the application of the CAD method in the context of *satisfiability modulo theories* (*SMT*) solving. A traditional SMT solver analyses the Boolean structure of a (usually quantifier-free first-order logic) formula by a SAT solver and uses one or more

theory solvers to check the consistency of (sets of) theory constraints. The Boolean and theory checks often proceed interlaced, i.e., a partial Boolean assignment is extended stepwise and for each extension the consistency of an increasing set of theory constraints needs to be checked.

Definition 2 (Extension and reduction)**.** *Let atom(φ) be the set of theory atoms contained in a first-order formula φ. We consider φ' an* extension *of φ if atom(φ) $\subset$ atom(φ'). Conversely, we call φ a* reduction *of φ'.*

To be able to efficiently exploit CAD in this context, we need an adaptation that is *incremental* in the sense that once it has found a satisfying solution for a set of input constraints it is able to re-use the results of those computations to check the consistency of an extended set of input constraints. Furthermore, if a conflict is detected either at the Boolean or at the theory level (meaning that the currently considered partial assignment cannot be extended to a satisfying solution) then backtracking at the Boolean level reduces the current partial assignment, thereby also reducing the set of constraints that needs to be checked for consistency; for these steps it is advantageous if the theory solver is also able to *backtrack* by removing constraints from its input set without forgetting all information about previous computation steps.

Depending on whether these abilities are available, we distinguish three settings: In a *non-incremental* setting, CAD is always used on input formulas in the traditional fashion. In contrast to that, a *forward incremental* CAD is able to re-use information to decide the satisfiability of a sequence of formulas where every formula is an extension of the previous one. The hope is to solve subsequent problems faster by retaining the intermediate results of the previous calculations. In the forward incremental setting, backtracking removes all knowledge about previous checks. When *backtracking incrementality* is considered, the solver is able to remove constraints from its input set without losing all information about previous computations.

Technically, after a conflict, the next theory check is only required after backtracking and a subsequent extension of the partial assignment. In this case the next input formula is neither a reduction nor an extension of the previous one. We consider this to be two steps, a reduction followed by an extension, though the intermediate CAD computation is not performed in practice.

For the CAD adaptation, the Boolean search is irrelevant and we therefore only consider the communication at its interface. Incrementality (backtracking) reduces to adding (and removing) theory atoms from this set. There are two fundamentally different approaches to backtracking: In *chronological* backtracking theory atoms are removed in reversed order as they were added, so that they essentially form a stack. In *non-chronological* backtracking theory atoms can be removed in any order. As non-chronological backtracking is more general but traditional SMT solvers use chronological removal, which allows for an easier data structure, we consider both scenarios.

3.2. Incrementality within CAD

We recall that we restricted the input formulas to our CAD to conjunctions of constraints with existentially quantified variables. Thus finding a single satisfying sample point witnesses the satisfiability of the (existentially quantified) input formula and we can prematurely stop any computation. At this point the CAD may not be fully computed, but a partially constructed CAD is sufficient to answer the posed question.

Assuming we manage to keep the CAD in some kind of consistent state we can resume the computation based on the partial information already computed. The purpose of resuming the

computation may be to find another satisfying sample or – as in our case – proceeding with a slightly changed input.

An *incremental* or *partial lifting* was already proposed in Collins and Hong (1991) and we borrow ideas from there (see Section 4). However, to generate a satisfying sample point we might not need the full projection. We develop two approaches to perform also the projection in an incremental fashion. Using *simple incremental projection* we consider the input polynomials one after another, compute the full projection and try to find a satisfying sample after every polynomial (see Section 6). In contrast, using *full incremental projection* we decompose the (traditionally atomic) projection into many small projection steps and perform these individually, possibly switching back to lifting between two projection steps (see Section 7).

4. Incrementality in Lifting

We start with the description of the lifting (or construction) phase which is heavily inspired by partial CAD due to Collins and Hong (1991). The idea is that if we use CAD for satisfiability checking then we can stop the decomposition once we have found a satisfying sample point.

Definition 3 (Lifting tree)**.** *A* lifting tree *is a tuple* $(T, value, liftedWith, rootOf, evaluation, Q)$ *with the following components:*

- T *is a tree either being empty or having a finite set* V *of nodes with a root node* $r \in V$ *and child relation* $children : V \to \mathcal{P}(V)$*;*
- $value : (V \setminus \{r\}) \to \mathcal{R}$ *assigns a real algebraic number to each node;*
- $liftedWith : (V \setminus \{r\}) \to \mathcal{P}(\mathbb{P})$ *assigns a set of polynomials to each node;*
- $rootOf : (V \setminus \{r\}) \to \mathcal{P}(\mathbb{P})$ *assigns a set of polynomials to each node;*
- $evaluation : (V \setminus \{r\}) \to (C \to \{F, T, U, ?\})$ *assigns evaluation results (false, true, unknown and not evaluated) of constraints to each node; and*
- Q *is a priority queue containing nodes from* V *and using some user-defined order.*

We construct for every node from V a sample point s in the following way: the root node is assigned the *empty sample point* of dimension zero, and every other node extends the sample point of its parent node by its associated value. We can obtain the sample point as a variable assignment using Algorithm 1.

We define the level $level(s)$ of the sample point of a node s by its dimension (which is also the length of the path from the root leading to the node, and also the level of the CAD at which it is computed). In the following we also use the term *sample point* for nodes of T, meaning their (partial) sample points, and write also $s.f$ for $f(s)$ (e.g., $s.level$) and $s.f := e$ for modifying the function f to assign e to s. The purpose of the remaining components will become clear in the further course of this section.

A lifting tree represents the lifting phase of the CAD where we start at the bottom with the root node (at level zero) and proceed upwards until we obtain (full-dimensional) sample points of level n. To lift a sample point we compute extensions for it as described in Section 2.2 and append these extended sample points as new children to the current sample point. With this data structure we are able to store the sample points in a tree instead of just enumerating them locally in a recursive fashion.

Algorithm 1: Generate a sample point for a node from L

Input : Node v from lifting tree L
Output: Sample point α_v

```
1 Function Assignment(v)
2     α_v := ()
3     while v ≠ root do
4         α_v := compose(v.value, α_v)
5         v := v.parent
6     return α_v
```

This view essentially translates the lifting phase into a tree search problem for a satisfying sample point on the topmost level. By globally storing the state we can easily pause and continue the search and always retain the current progress. Rather than using a fixed search order (such as depth-first or breadth-first), we aim for more flexibility and use the priority queue Q to impose a user-defined order on the sample points that are still to be lifted. As we discontinue the search as soon as a satisfying sample is found, the state of the search is not local to a single CAD computation. On the contrary the lifting tree including the sample queue is persistently stored and updated throughout a whole sequence of CAD executions as presented in Section 3.

As a refinement compared to the lifting phase of the regular CAD, we allow the lifting to lift a sample point only with a single polynomial instead of all polynomials from the respective level. This allows us to defer work – lifting with respect to *difficult* polynomials – in the hope that using *easy* polynomials already guide us towards a satisfying sample point.

In order to remember which sample point was already lifted with respect to which polynomial, we store a set of polynomials for every sample point using *liftedWith* (see Section 10.11 for some details about the actual implementation). Whenever we lift a sample point s we select a polynomial $p \notin s.liftedWith$ and add p to $s.liftedWith$ afterwards. Note that this implies that all roots of p under s are considered in one go.

For the purpose of backtracking in Section 8 we store the set of all polynomials that vanish at a sample point s in $s.rootOf$. We call a sample point s a *root sample* if $s.rootOf \neq \emptyset$, otherwise a *non-root sample*. We also write $s.isRoot$ as a shortcut for $s.rootOf \neq \emptyset$. Intuitively the polynomials in $s.rootOf$ are the *reasons* that this particular sample point exists.

Algorithm 2: Merge two lists of samples

Input : Lists of samples S_1, S_2 from the same level with
$s, s' \in S_i \to s.value \neq s'.value$ for $i = 1, 2$
Output: Merged list of samples containing S_1, S_2

```
1 Function MergeSamples(S_1, S_2)
2     R := ∅
3     for s_1 ∈ S_1, s_2 ∈ S_2 with s_1.value = s_2.value do
4         Add sample(s_1.value, s_1.rootOf ∪ s_2.rootOf) to R
5         Remove s_1 from S_1 and s_2 from S_2
6     return sorted(R ∪ S_1 ∪ S_2)
```

During the lifting of a sample point s with a polynomial p it may be the case that s already has some child nodes and we need to merge the set of newly obtained sample points with the existing ones. This procedure decomposes into three separate steps: finding the roots of p under s, merging these roots into the child nodes that are already present and finally adding missing sample points. We assume that a procedure *roots*() exists and show the other two steps in Algorithms 2 and 3. Algorithm 2 merges two lists of samples and orders them by the value of the samples. It unifies samples whose values are identical and makes sure that the unified sample is a root sample if at least one of the original samples was a root sample.

Algorithm 3: Lift sample point with a single polynomial

Input : Lifting tree L, sample point s of some level i, polynomial p of level $i+1$
Output: Updated lifting tree L

```
1  Function LiftSample(L, s, p)
2      R := ∅
3      for r ∈ roots(p(s)) do
4          Add sample(value = r, rootOf = {p}) to R
5      S := MergeSamples (L.children(s), R)                    // Algorithm 2
6      if head(S).isRoot then
7          prepend s with s < head(S).value to S
8      if tail(S).isRoot then
9          append s with s > tail(S).value to S
10     for every two consecutive samples s_i, s_{i+1} do
11         if s_i.isRoot and s_{i+1}.isRoot then
12             value := value from (s_i.value, s_{i+1}.value)
13             insert sample(value, rootOf = ∅) between s_i and s_{i+1}
14     add S \ L.children(s) to L.Q
15     replace L.children(s) by S
```

Algorithm 3 first computes all real roots of the polynomial p under s and merges this list of root samples into the list of existing children of s by calling Algorithm 2. The remainder of Algorithm 3 completes this merged list S such that a non-root sample point exists before the smallest root sample, in between every two consecutive root samples and above the largest root sample. This implements the lifting drafted in Section 2.2 in an incremental fashion.

Before actually using Algorithm 3, we can evaluate a sample point on the input constraints to avoid lifting unsatisfiable samples in the spirit of Collins and Hong (1991). The evaluation result can be true or false (denoted by T and F), but also undefined or unknown (denoted by U and ?) if the constraint contains additional variables or the evaluation was simply not done yet. As this check is performed frequently, we store the evaluation results in *s.evaluation* to avoid evaluating the same sample on the same constraint over and over again.

Whenever a constraint evaluates to false on some sample point, Algorithm 3 is not called and the sample point is skipped. The process of this evaluation is shown in Algorithm 4. This technique is particularly valuable if constraints with only a few variables are present that conflict with a large number of sample points. The practical impact can be very significant, as for example shown by Loup et al. (2013).

Algorithm 4: Evaluate sample on constraints

Input : Sample point s, Constraints C
Output: The evaluation result of C over s, either T (true), F (false) or U (unknown)

```
1 Function Evaluate(s, C)
2     if s.evaluation(c) = F for some c ∈ C then return F
3     while s.evaluation(c) =? for some c ∈ C do
4         s.evaluation(c) := evaluate(s, c)
5         if s.evaluation(c) = F then return F
6     if s.level = n with n being the dimension of the CAD then return T
7     return U
```

The complete lifting process is shown in Algorithm 5, assuming polynomials $P \subseteq \mathbb{P}$. We later see that certain samples have to be reconsidered after we paused the lifting, for example if new polynomials were added to the projection. This incurs a certain overhead even if no further children are computed from the samples we reconsider, but it is usually very small.

Algorithm 5: Lifting

Input : Lifting tree L, polynomials P
Output: (SAT, s) with a satisfying sample s, or $UNSAT$

```
1  Function Lifting(L, P)
2      while not L.Q.empty do
3          s := L.Q.top
4          switch Evaluate (s,C) do                    // Algorithm 4
5              case T do return (SAT, s)
6              case F do L.Q.remove(s), continue
7          if s.liftedWith = P_{s.level+1} then
8              L.Q.remove(s)
9          Select p ∈ P_{s.level+1} \ s.liftedWith
10         LiftSample (L, s, p)                        // Algorithm 3
11         Add p to s.liftedWith
12     return UNSAT
```

5. Data Structure for Projection

The projection is commonly seen as a single procedure that is performed as a whole before even starting the lifting process. The projection can however be split into many individual steps and we show how to do this using the example of McCallum's projection operator that we already presented in Definition 1. Given a set of input polynomials $I \subseteq \mathbb{P}$ we accumulate the set of all projection polynomials P by applying this rule recursively until we have used $Proj_x$ for all variables but x_1. We give a possible implementation for McCallums's projection operator as a first algorithmic description in Algorithm 6.

Algorithm 6: Compute the set of all projection polynomials

Input : Set of input polynomials I, variables X
Output: Set of all projection polynomials P

```
1 Function Projection(I)
2     P := I
3     for i = n, n − 1, ..., 2 do
4         for p ∈ P_i do
5             P := P ∪ {disc_{x_i}(p)} ∪ coeffs_{x_i}(p)
6         for p, q ∈ P_i do
7             P := P ∪ {res_{x_i}(p, q)}
8     return P
```

We consider the computations in Line 5 and Line 7, though depending on the computations of earlier iterations, as individual *steps*. Notably, we distinguish between *single projection steps* (in Line 5) that process a single polynomial and *paired projection steps* (in Line 7) that combine two polynomials. We choose this level of abstraction because it is common to all prominent projection operators.

To obtain an incremental version of this overall projection process we adopt a different perspective. We consider the projection to be a directed graph where every node corresponds to a polynomial from P and is associated to a level i that corresponds to the variable x_i.

Definition 4 (Projection data structure)**.** *Let $PROJ = (P, E)$ be a directed acyclic hypergraph with polynomials P acting as vertices and hyperedges $E \subset (P \cup (P \times P)) \times P$. We associate a* level *i with every polynomial $p \in P$ which is the index of its largest variable x_i. A hyperedge $e \in E$ has one or two* source *polynomials and a single* target *polynomial and the level of all source polynomials is always larger than the level of the target polynomial.*

We call $PROJ$ a projection data structure*. We write $PROJ_i$ for the set of all polynomials from $PROJ$ on level i.*

The intention of this definition is that every hyperedge is directly associated with one of the steps from Algorithm 6. A hyperedge $e \in P \times P$ stems from a *single projection step* while $e \in (P \times P) \times P$ originates from a *paired projection step*. Similar to the lifting tree, this projection data structure allows us to suspend the computation of the projection at any time and resume without the need to recompute anything.

This computation can be taken as an iterative refinement of the CAD. Every step possibly yields new polynomials that separate existing regions into multiple smaller ones. Though these intermediate partially refined states bear no strict mathematical meaning per se, other than being a precursor to a CAD, they may be enough to determine satisfiability. Each separation of regions forces the lifting process to construct more sample points which may just hit a satisfying region – at which point we can stop immediately.

We can also interpret the projection data structure as a dependency graph that models the data flow within Algorithm 6 and we can reorder the individual steps arbitrarily, as long as we adhere to the partial order implied by the dependency graph.

Note that multiple steps can produce the same polynomial (possibly after considering normalization and taking care of the restrictions of the projection operator discussed above) in a

lower level which manifests in *PROJ* as multiple hyperedges with the same target polynomial. In this case polynomials are not inserted a second time and thus do not induce new projection steps. We thus do not know the graph structure of *PROJ* beforehand: it rather evolves during the computation and the steps that still need to be computed are expanded with every new polynomial.

It is important to note that we do not actually implement *PROJ* as defined above but merely use it as a mental model. As long as we do not backtrack the hyperedges are not needed and therefore not stored. For backtracking *PROJ* is extended by so-called *origins* in Section 8 that partially represent the hyperedges.

Algorithm 7: CAD without incrementality

Input : List of constraints C
Output: (SAT, s) with a satisfying sample s, or $UNSAT$

```
1 Function CAD_NN(PROJ, L)
2     PROJ := Projection(PROJ.Q)          // Algorithm 6
3     return Lifting(L, PROJ)             // Algorithm 5
4 PROJ := ∅
5 PROJ.Q := polynomials(C)
6 L := empty lifting tree
7 return CAD_NN(PROJ, L)
```

The presented methods and data structures can already be used to compute a CAD in the regular non-incremental fashion as shown in Algorithm 7. We present different versions of these when the computation may be suspended and how we reorder the individual steps in Section 6 and Section 7. Before diving into how to build an incremental projection we give an example of Algorithm 7.

Example 5. *We extend the previous example from Figure 3 with another constraint to* $x_1^2 + x_2^2 < 4 \wedge x_1 \cdot x_2 > 1 \wedge x_1^2 + x_2 \leq 3$ *as shown in Figure 5.*
We start computing the full projection from $\{x_1^2 + x_2^2 - 4, x_1 \cdot x_2 - 1, x_1^2 + x_2 - 3\}$ *by calling* `Projection`*. For each of these polynomials we consider the discriminant and the coefficients as well as the resultant for all pairs of polynomials. This results in the five univariate polynomials* $\{x_1, x_1^2-4, x_1^3-3x+1, x_1^4-4x_1^2+1, x_1^4-5x_1^2+5\}$*. We continue with* `Lifting` *using these polynomials and eventually find a satisfying sample from one of the* 225 *regions.*

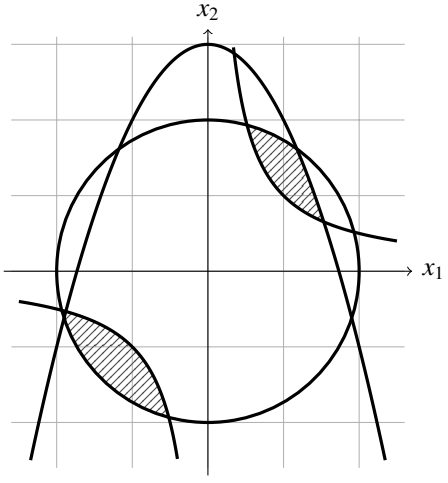


Figure 5: Example for the input formula $x_1^2 + x_2^2 < 4 \wedge x_1 \cdot x_2 > 1 \wedge x_1^2 + x_2 \leq 3$

6. Simple Forward Incremental Projection

We start with a simple version of incrementality to illustrate the general idea. For this approach that we call *simple incremental* we maintain a queue of input polynomials and extend *PROJ* with one polynomial at a time. We denote this queue by *PROJ.Q*.

Algorithm 8: Extend the projection with a new polynomial

Input : Existing projection *PROJ*, new polynomial r
Output: Levels on which polynomials were added

```
1  Function ExtendProjection_S(PROJ, r)
2      Q := {r}
3      levels := ∅
4      for i = n, n − 1, ..., 1 do
5          if Q_i = ∅ then
6              continue
7          for q ∈ Q_i do
8              Q := Q ∪ {disc_{x_i}(q)} ∪ coeffs_{x_i}(q)
9              for p ∈ PROJ_i do
10                 Q := Q ∪ {res_{x_i}(p, q)}
11         Q := Q \ PROJ
12         Add i to levels
13         Add Q_i to PROJ
14     return levels
```

Whenever we add a polynomial r to *PROJ*, we extend *PROJ* as shown in Algorithm 8. This procedure extends the existing projection *PROJ* by performing all computation steps that in some way depend on the newly added polynomial r. After all polynomials have been added by Algorithm 8 *PROJ* contains the same data as if it had been computed by Algorithm 6 on all polynomials in the first place. Note that we treat *PROJ* as a mutable argument in Algorithm 8 (and Algorithm 10) and the actual extension of the projection happens as a side effect of the function call while the return value merely indicates which levels have been modified.

This gives us a way to start with the empty projection and iteratively refining the projection with one input polynomial at a time. We suspend the projection process after each input polynomial and go back to the lifting process, always hoping that we can avoid the rest of the projection if the lifting can already find a satisfying sample.

The process is detailed in Algorithm 9 which is a complete CAD method for satisfiability problems. Note that we implicitly choose an order on the constraints in Line 11 hoping that we can avoid looking at some constraints altogether. However the granularity of work that we can avoid is extending the projection by a whole constraint at a time. We show how to improve on this by allowing for smaller steps in the next section.

Example 6. *We reconsider Example 5 and assume that the polynomials are considered in the order* $x_1 \cdot x_2 - 1, x_1^2 + x_2 - 3, x_1^2 + x_2^2 - 4$. *Note that this ordering is the result of using the total degree as a sorting criterion.*

Algorithm 9: CAD with simple forward incrementality

Input : List of constraints C
Output: (SAT, s) with a satisfying sample s, or $UNSAT$

Function $\texttt{CAD}_{SF}(PROJ, L)$
  while *not* $PROJ.Q.empty$ **do**
    $res := \texttt{Lifting}(PROJ)$ `// Algorithm 5`
    if $res = (SAT, s)$ **then**
      return (SAT, s)
    $levels := \texttt{ExtendProjection}_S(PROJ, PROJ.Q.top)$ `// Algorithm 8`
    Add $\{s \in L \mid s.level + 1 \in levels\}$ to $L.Q$
    $PROJ.Q.pop()$
  return $\texttt{Lifting}(L, PROJ)$ `// Algorithm 5`
$PROJ := \emptyset$
$PROJ.Q := polynomials(C)$
$L :=$ empty lifting tree
$L.Q :=$ sort $nodes(L)$ heuristically
return $\texttt{CAD}_{SF}(PROJ, L)$

We start by calling `Lifting` *on an empty projection which typically guesses* $(0, 0)$ *and thus fails. Afterwards the projection is extended by the first polynomial* $x_1 \cdot x_2 - 1$ *and we call* `Lifting` *again. At this point the CAD looks like shown in Figure 6a. Note that SMT-RAT already tries to use* $x_1 = 1$ *as a sample but fails to find a satisfying value for* x_2 *based on the partial projection as it tries to extend it with* $x_2 = 0$, $x_2 = 1$ *and* $x_2 = 2$.

We continue by adding $x_1^2 + x_2 - 3$ *to the projection and arrive at Figure 6b. Which sample points are chosen in* `Lifting` *of course depends on a few heuristics, but SMT-RAT tries* $(1, 1.5)$ *at this point and finds a satisfying sample point before the third constraint is considered at all.*

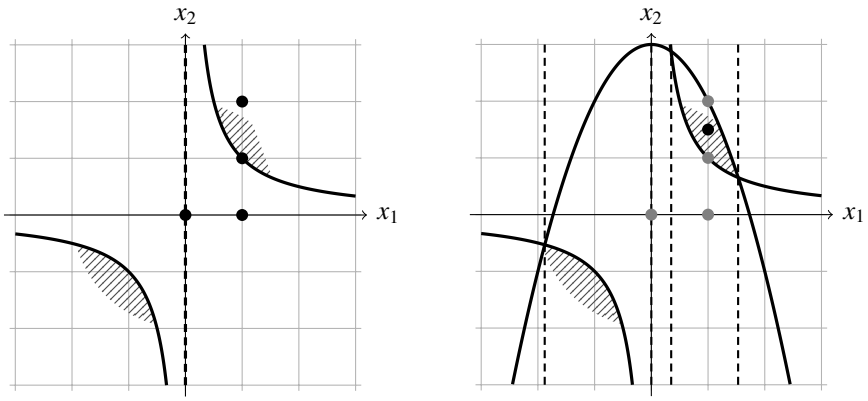
7. Full Forward Incremental Projection

To maximize the amount of work we may avoid the size of the individual steps performed during the projection should be reduced. We do that by considering every step from Algorithm 6 individually. We consider *projection candidates* that represent a single projection step that is to be performed within a projection.

Definition 7 (Projection candidate)**.** *We define* $PC = P \cup (P \times P)$ *to be the set of* projection candidates. *We call* $pc_p^i \in P$ single *and* $pc_{p,q}^i \in P \times P$ paired *projection candidates on level* i. *A projection candidate can be* evaluated *by a projection operator* $Proj$, *for example as follows using McCallum's operator from Definition 1:*

$$Proj(pc_p^i) = \{\text{disc}_{x_i}(p)\} \cup \text{coeffs}_{x_i}(p)$$
$$Proj(pc_{p,q}^i) = \{\text{res}_{x_i}(p, q)\}$$

We define $PC_p = \{pc \in PC \mid p \in Proj(pc)\}$ *as the set of projection candidates that* yield *a polynomial* p.



(a) Added $x_1 \cdot x_2 - 1$ (b) Added $x_1 \cdot x_2 - 1$ and $x_1^2 + x_2 - 3$

Figure 6: Simple incremental computation for $x_1 \cdot x_2 > 1 \wedge x_1^2 + x_2 \leq 3$

As we want to be able to suspend the computation after every such step, we maintain a list of *projection candidates* that still have to be performed that we call the *projection queue*. Whenever a polynomial is inserted into the projection, all projection candidates that are induced by the new polynomial are added to the queue. Again we denote this queue by *PROJ.Q*.

Algorithm 10: Extend the projection with a new polynomial

Input : Existing projection *PROJ*
Output: Levels on which polynomials were added

```
1  Function ExtendProjection_F(PROJ)
2      pc := PROJ.Q.top
3      PROJ.Q.pop()
4      levels := ∅
5      for p ∈ Proj(pc) do
6          if p ∈ PROJ then
7              continue
8          Let i such that p ∈ ℙ_i \ ℙ_{i−1}
9          Add pc^i_p to PROJ.Q
10         for q ∈ PROJ_i do
11             Add pc^i_{p,q} to PROJ.Q
12         Add p to PROJ_i
13         Add i to levels
14     return levels
```

This operation is shown in Algorithm 10. We use Definition 7 to obtain a set of polynomials that is the result of the respective projection step. For every polynomial of this set that is not already part of the projection, we create the induced projection candidates in Line 9 and Line 11 and finally add it to *PROJ*.

We could suspend the computation after every additional polynomial instead of the complete processing of a projection candidate. This however requires even more overhead to store these polynomials while the computation is suspended.

The order in which the projection candidates are processed is heuristic and we have plenty of properties that can be considered here. We could favor polynomials of smaller degrees or fewer variables, hoping that the projection operations are less costly. Using the presented projection operator a paired projection candidate only ever adds a single new polynomial. Meanwhile a single projection candidate adds the coefficients of the polynomial that however usually have a smaller degree.

Note that compared to the simple incremental version the projection queue *PROJ.Q* contains projection candidates instead of polynomials. As projection candidates refer to polynomials from *PROJ* we initially need to insert input polynomials into *PROJ* before we can start building any projection candidates from them. This observation has an interesting consequence for the lifting process: as the lifting happens with respect to all existing projection polynomials, the input polynomials are immediately *visible* to the lifting process which may be a significant overhead that we may want to avoid. We therefore introduce a *virtual level* $PROJ_\infty$ into *PROJ* that contains all input constraints and add artificial projection candidates pc_c^∞ to *PROJ.Q* that are evaluated using $Proj(pc_c^\infty) = \{p \mid c = p \sim 0\}$. This virtual level essentially mimics the behavior of Algorithm 9, and we finally obtain Algorithm 11.

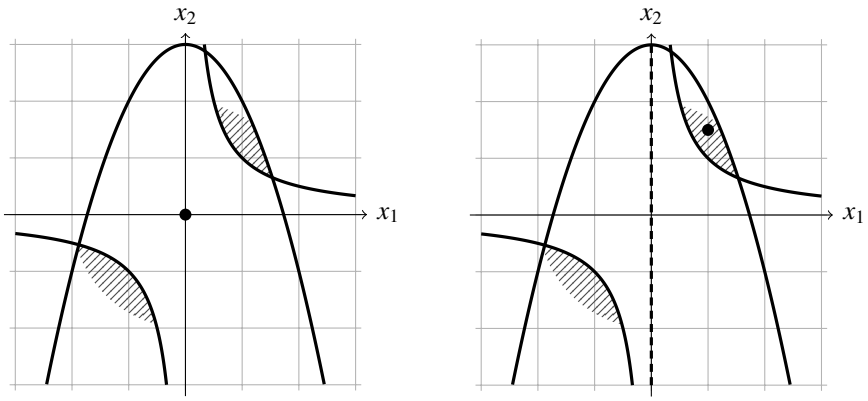
Algorithm 11: CAD with full forward incrementality

Input : List of constraints C
Output: (SAT, s) with a satisfying sample s, or $UNSAT$

Function $\texttt{CAD}_{FF}$ *(PROJ, L)*
  while *not PROJ.Q.empty* **do**
    $res :=$ `Lifting` $(L, PROJ)$ `// Algorithm 5`
    if $res = (SAT, s)$ **then**
      return (SAT, s)
    $levels :=$ $\texttt{ExtendProjection}_F$ $(PROJ)$ `// Algorithm 10`
    Add $\{s \in L \mid s.level + 1 \in levels\}$ to $L.Q$
  return `Lifting` *(L, PROJ)* `// Algorithm 5`
$PROJ := \emptyset$
Initialize $PROJ_\infty$ with C
$PROJ.Q := \{pc_c^\infty$ for every $c \in C\}$
$L :=$ empty lifting tree
$L.Q :=$ sort $nodes(L)$ heuristically
return $\texttt{CAD}_{FF}$ *(PROJ, L)*

Example 8. *We again have a look at our input problem from Example 5. We assume that we first considered* $x_1 \cdot x_2 - 1$*, then* $x_1^2 + x_2 - 3$ *and continue with the* single *projection steps of these two.*

As before we start by calling `Lifting` *on an empty projection, guess* $(0, 0)$ *and fail. We now add the first polynomial* $x_1 \cdot x_2 - 1$ *to the projection but do not evaluate the consequential projection candidates yet. The subsequent call to* `Lifting` *therefore finds no new samples and fails again. We try to add the second polynomial* $x_1^2 + x_2 - 3$*, again without evaluating the resulting projection candidates, yielding the situation shown in Figure 7a.*



(a) Added $x_1 \cdot x_2 - 1$ and $x_1^2 + x_2 - 3$ (b) Added the coefficient x_1

Figure 7: Fully incremental computation for $x_1 \cdot x_2 > 1 \wedge x_1^2 + x_2 \leq 3$

Finally we perform a first real projection step by computing the single *projection of* $x_1 \cdot x_2 - 1$ *which only results in* x_1 *as the discriminant is constant. Due to the new root at* $x_1 = 0$ *SMT-RAT also checks the sample points* $x_1 = -1$ *and* $x_1 = 1$ *and* `Lifting` *finds the satisfying sample shown in Figure 7b.*

8. Backtracking

The incrementality presented up to this point allows us to lazily extend our CAD computation. We know how to add new constraints and try to find a satisfying sample point for this extended problem without computing the whole CAD. In SMT solving, we also want to do the reverse: removing constraints from our partial CAD to explore a different branch in our Boolean search. This implies that we want to remove all polynomials from *PROJ* that originate from a constraint that is removed.

We first present a simple variant of this removal that we call *chronological backtracking*. The assumption is that constraints are added in a particular order and only the last constraint can be removed. We can regard the set of constraints as a *stack* where new constraints are added on top and only the topmost element can be removed. This restriction allows for a rather easy scheme to figure out what to remove from the projection.

Afterwards we consider the general case that we call *non-chronological backtracking* where any constraint can be removed at any time. This gives us some freedom to avoid removals in certain cases, but introduces some overhead due to more involved bookkeeping.

Example 9 (Different backtracking strategies)**.** *Let* c_1, c_2, c_3 *be a set of constraints. Assume we first add* c_1 *and subsequently* c_2 *to our constraint set and computed the CAD. If we now want to compute the CAD for* c_2, c_3 *we have two possibilities: either we remove* c_2 *and* c_1 *and then add* c_2 *and* c_3 *or we only remove* c_1 *and add* c_3*.*

In the first case the constraints are removed chronologically *while the removal of* c_1 *is* non-chronological *in the second case. We need two removals and two additions in the first case compared to one addition and one removal in the second case.*

For any backtracking attempt, we need to know *a reason* for some object to be present. Once this reason ceases to exist we can remove this object.

In our projection there may be more than one reason for a single polynomial because multiple projection candidates may produce the same polynomial (or different polynomials with a common factor). This means that we have not only a single but several reasons that we call *origins* and a polynomial can only be removed when no origin is left. We define what exactly *origins* are for chronological and non-chronological backtracking separately.

Note that we assume that auxiliary data structures are always cleaned appropriately if a constraint or a polynomial is removed from the projection. This includes the queues in Algorithm 9 and Algorithm 11, but also the lifting tree L as explained in Section 8.3. Furthermore we assume the case of *full incrementality* but both approaches can be applied directly to the other variants as well.

8.1. Chronological Backtracking

In the case of chronological backtracking we have a strict ordering on the input constraints. We define *the origin* of a polynomial to be the smallest constraint that yielded the polynomial.

Definition 10 (Origin for chronological backtracking)**.** *Let $p \in PROJ$ and PC the set of projection candidates. We define* the origin of a projection candidate *as*

$$\begin{aligned}\sigma(pc_p) &= origin(p)\\ \sigma(pc_{p,q}) &= \max\{origin(p), origin(q)\}\\ \sigma(pc_c^{\infty}) &= c\end{aligned}$$

and the origin of a polynomial *as*

$$origin(p) = \min\{\sigma(pc) \mid pc \in PC_p\}.$$

Following this definition it suffices to store a single input constraint for every polynomial. Note that the origin of a polynomial may change during the computation as new projection candidates are executed. This definition provides an easy criterion for when to remove a polynomial from *PROJ*: if an input constraint is removed all polynomials that have this constraint as their origin can also be removed. This immediately yields Algorithm 12 which modifies its (mutable) argument *PROJ* accordingly.

Algorithm 12: Remove constraint using chronological backtracking

Input : Projection *PROJ*, input constraint c (that was added last)

1 **Function** $\texttt{Remove}_C(PROJ, c)$

2 $\quad P := \{q \in PROJ \mid origin(q) = c\}$

3 $\quad$ Remove P from *PROJ*

Note that we can use a projection data structure that only supports *chronological* backtracking within a system that backtracks *non-chronologically*, essentially dropping the condition that c should be the input constraint that was added last. Given the ordered list of constraints C Algorithm 13 serves as an adapter for this case. When removing an older constraint it removes all newer constraints and re-adds them to the projection queue afterwards. This may incur a significant overhead but it happens only rarely in practice.

Algorithm 13: Transform non-chronological to chronological backtracking

```
Input : Projection PROJ, any input constraint c, ordered list of constraints C
1 Function Remove(PROJ, c, C)
2     for c' ∈ reversed(C) do
3         Remove_C (PROJ, c')                                  // Algorithm 12
4         if c = c' then
5             return
6         Add c' to PROJ.Q
```

8.2. Non-chronological Backtracking

If we allow non-chronological backtracking it is not sufficient to store a single constraint. Instead we store a set of *origins* consisting of projection candidates which essentially represent the hyperedges of $PROJ$.

Definition 11 (Origins for non-chronological backtracking)**.** *Let $p \in PROJ$ and PC be the set of projection candidates. We define* the origins *of a polynomial as $origins(p) = PC_p$ and call $o \in origins(p)$ an* origin of p.

We assume that the implementation makes sure that the origins of any polynomial are always *reduced*, meaning that it removes an origin if the respective input constraint or a referenced polynomial is removed from $PROJ$. Whenever the origins of some polynomial p become empty, either because an input constraint or some other polynomials were removed, p can be removed from $PROJ$. This gives us a criterion to remove polynomials from $PROJ$ after some input constraint has been removed that immediately yields Algorithm 14.

Algorithm 14: Remove constraint from projection

```
Input : Projection PROJ, any input constraint c
1 Function Remove_N(PROJ, c)
2     R := {c}
3     for i = n, ..., 1 do
4         Remove R from origins(p) for all p ∈ PROJ_i
5         P := {p ∈ PROJ_i | origins(p) = ∅}
6         Remove P from PROJ_i
7         R := R ∪ P
```

8.3. Backtracking in Lifting

To avoid unnecessary work we also remove obsolete sample points from the lifting tree L when polynomials are removed from the projection. Our aim is to clean up L so that after removing input constraints it does not contain any sample points that are superfluous with respect to the remaining input constraints. We first define which sample points are obsolete and then give a mechanism to remove exactly these sample points from L.

We observe that there are two types of sample points that are fundamentally different when we are concerned with backtracking. A root sample s (as described in Section 4, representing a root of one or more projection polynomials) exists due to the projection polynomials in $s.rootOf$ and can therefore only be removed if $s.rootOf$ becomes empty. A non-root sample on the other hand exists because of the neighboring root samples. Thus whenever a root sample is removed we also remove either of its two neighbors.

Algorithm 15: Backtrack the lifting tree

```
   Input  : Lifting tree L, projection PROJ
 1 Function BacktrackLifting(L, PROJ)
 2     for s ∈ L with s.rootOf ≠ ∅ do
 3         s.rootOf := s.rootOf ∩ PROJ
 4         if s.rootOf = ∅ then
 5             t := some neighboring non-root sample of s
 6             Remove subtrees of s and t from L
```

Algorithm 15 uses exactly this insight to remove obsolete sample points. For every sample point s it removes all polynomials from $s.rootOf$ that are no longer part of $PROJ$. If this results in $s.rootOf$ being empty, s and a neighbor of s is removed including their whole subtrees.

Note that for the case of simple incrementality, we do not actually need to store the full set of polynomials in $s.rootOf$. Instead we can reduce this set to the smallest constraint involved, very similar to the origins of the polynomials.

9. The Overall Procedure

We can now combine all the methods we have seen into a CAD object that is able to work incrementally and may backtrack. This *CAD object* as shown in Algorithm 16 can for example be used as a theory solving module in an SMT solver where we use any of the Algorithms 7, 9 or 11 within the `Check` method.

Algorithm 16: CAD object

```
 1 object CAD
 2     Lifting tree L
 3     Projection PROJ
 4     Function Add(Constraint c)
 5         Add c to PROJ.Q
 6     Function Remove(Constraint c)
 7         Remove (PROJ, c)                      // Algorithm 12, 13 or 14
 8         BacktrackLifting (L, PROJ)                      // Algorithm 15
 9     Function Check()
10         return CAD(PROJ, L)                    // Algorithm 7, 9 or 11
```

10. Further Topics

Since the proposal of CAD by Collins (1975) an abundance of optimizations and extensions has been accumulated that we have not mentioned yet. Naturally we want to integrate these existing techniques with our proposed ones wherever suitable. In the following we briefly discuss some of these contributions and indicate how they affect or interact with our approach.

10.1. Additional Information on Polynomials

Some modifications to the CAD require additional data to be stored for polynomials or sample points. A prominent example is the projection operator due to Brown (2001) which tags polynomials with the information whether they are sign-invariant or order-invariant. Such additional data can easily be added to *PROJ* as an additional label to the polynomials.

10.2. Variable Order

We assumed a total variable ordering on our variables $x_1, \ldots, x_n$. Choosing a *good ordering* is a topic of active research, for example in Dolzmann et al. (2004) or Huang et al. (2014) – and it can have a huge impact on whether a specific problem is practically solvable. In our implementation we use the *triangular heuristic* as described by England et al. (2014) with respect to the set of all polynomials of the whole input for all experiments. We refrain from discussing the impact of different variable orderings in our use case here as we only performed limited experiments with inconclusive results yet, but plan to conduct further research on this issue.

10.3. Forwarding Polynomials

Whenever a projection step using polynomials from level i is performed we insert the result to level $i - 1$. In the special case of a resulting polynomial that does not contain x_{i-1} however, we can immediately forward it to a lower level. This essentially skips some projection steps that only produce redundant polynomials anyway.

We argue that this optimization is sound using the example of McCallum's projection operator by showing that no relevant polynomial factor is produced for such a polynomial. We defined it as

$$Proj_x(P) = \bigcup_{p \in P} \text{coeffs}_x(p) \cup \{\text{disc}_x(p) \mid p \in P\} \cup \{\text{res}_x(p, q) \mid p, q \in P\}$$

For a polynomial p that does not contain x we have $\text{coeffs}_x(p) = \{p\}$ which effectively only forwards p to the next level. The discriminant of p is defined as the resultant of p and its derivative p'. With respect to x we have $p' = 0$ and thereby $\text{disc}_x(p) = 0$. For the resultant of p and some other polynomial q it is known that $\text{res}_x(p, q) = p^e$ with e being the degree of q. Hence p only adds powers of itself to the next level which can be reduced to p. Thus computing all these projection steps is equivalent to simply forwarding p in the first place.

10.4. Discarding Polynomials

We observe that we can safely discard polynomials that have no real roots, for example constants or polynomials like $x^2 + 1$. If we have a bounded problem – that is a variable x_i must be from an interval I_i that we call the *bounds of* x_i – we can extend this notion: a polynomial can be discarded if it has no real roots *within the given bounds*. We refer to Loup et al. (2013) for more details on how to identify such polynomials.

We extract bounds from input constraints of the form $x_i + r \sim 0$ with $r \in \mathbb{R}$ and thus these bounds may change when constraints are added or removed. This implies that whenever a constraints is removed from our CAD we must check whether this changes any of the bounds and potentially recheck all polynomials that we discarded.

To do this we maintain a list of polynomials $P_\times$ that do not vanish within the current bounds and a list of projection candidates $PC_\times$ that are based on these not vanishing polynomials. When a polynomial is added to the projection we check whether it vanishes within the bounds. If it does not, we add it to $P_\times$ instead of *PROJ* and add the respective projection candidates to $PC_\times$ instead of *PROJ.Q*. Whenever a bound is removed we reevaluate all polynomials from $P_\times$ and move all polynomials that now vanish to *PROJ* including the associated projection candidates.

Note that whenever a new bound is added polynomials already part of *PROJ* could be discarded although they may already have produced other polynomials. We could move the discarded polynomial to $P_\times$ and remove all polynomials originating from it from *PROJ*, leading to a large number of recomputations when the bound is removed again. Alternatively we could store these dependent polynomials alongside with the not vanishing one in $P_\times$, incurring a large bookkeeping overhead. Our implementation does not perform this check at all but keeps these polynomials in *PROJ*. Though this may lead to larger projections than necessary we suspect that the overhead of any of the two solutions is significant. We however plan to study this topic in the future.

10.5. Polynomial Factors

A common technique is to factorize all polynomials and consider all factors individually. This immediately eliminates all multiple factors, can reduce the polynomial degree which is one of the main drivers of computational complexity in the CAD and also yields a *square-free basis* that we need for some projection operators anyway. This can be implemented (almost) transparently by returning *the irreducible factors* of input polynomials as well as coeffs_x, disc_x and res_x instead of the (unfactorized) input polynomials, coefficients, discriminants and resultants, requiring a slight change in notation in Algorithm 6 and Algorithm 8.

We acknowledge that a square-free basis (and the contents) can be coarser than that but requires a *global view* on all polynomials to be computed or might need to change earlier polynomials when new ones are added, making it hard to integrate with the incremental projection described above. Fully factorizing polynomials avoids this problem and can be applied to every polynomial (and every projection step) individually. In this sense factorization is not an optimization but rather an integral part in ensuring soundness if a projection operator is used that requires a square-free basis.

Our approach allows for this technique and we simply use the origin of the polynomial for all factors to ensure proper backtracking. Note that if multiple polynomials have common factors this may reduce the number of polynomials that are removed during backtracking and thereby may further strengthen our proposed approaches.

10.6. Delineating Polynomials

The usage of some projection operators, most notably the ones due to McCallum (1998) and Brown (2001), may lead into problems during the lifting phase. Some of these cases can be rectified by adding *delineating polynomials* to the projection, though the CAD using these projection operators is incomplete in general. This may be a problem for regular implementations of CAD as it requires going back to the projection phase and potentially challenges architectural assumptions of the implementation.

It however neatly integrates into the proposed adaption of CAD as we regularly switch between projection and lifting anyway. A *delineating polynomial* is added for a polynomial p if p vanishes identically zero over a whole region and thus is a direct consequence of p. The origins of this newly added delineating polynomial can thus be copied from p, integrating this method nicely into our backtracking mechanism.

10.7. Explanations for Unsatisfiability

A common extension, especially when used in the context of SMT solving, is the generation of explanations for unsatisfiability or *infeasible subsets*. We consider an *infeasible subset* a subset of the input constraints that are inconsistent over the reals. It is usually beneficial to spend some effort to find a small set as this can provide major performance improvements during the overall solving process.

A common approach for the CAD as for example presented in Jaroschek et al. (2015) or Hentze (2017) relies on the leaf sample points of the lifting tree. The fundamental reasoning is as follows: given that the set of constraints is infeasible, every leaf sample point conflicts with at least a single constraint. Finding a small set of constraints such that every leaf sample point conflicts with at least one of these constraints translates to a set cover problem. This approach is independent of the inner workings of the CAD as long as we can extract the leaf sample points.

While Jaroschek et al. (2015) is more theoretical and also explores its application beyond CAD, Hentze (2017) provides more details on an efficient implementation and concrete heuristics. Note that although Jaroschek et al. (2015) presents the problem as a linear program, both implementations actually consider the set cover encoding from Hentze (2017) and a standard greedy heuristic as the main selection strategy. An important contribution of Hentze (2017) is the realization that most instances we face when solving the standard benchmarks from Barrett et al. (2016) can actually be solved optimally after appropriate preprocessing. The implementation described in Hentze (2017) is part of SMT-RAT and thus used in the benchmarks below.

10.8. Usage as a general CAD Method

Though we have gone to great lengths to provide an CAD method that is optimized for satisfiability questions, it can easily be used as a general purpose CAD as well, for example for general quantifier elimination. This already proved to be valuable as quantified problems are now being considered in the SMT community as well, witnessed for example by a (quantified) `NRA` category in Barrett et al. (2016), and we successfully used this implementation without many changes for quantifier elimination in Neuhäuser (2018). The proposed algorithms change only two things, compared to the general CAD. Sample points are not lifted further if they are found to conflict with some constraint and we terminate as soon as a satisfying sample is found.

We can easily disable both if we skip the evaluation of sample points in Line 4 of Algorithm 5. If we do this, `Lifting` always returns $UNSAT$ and our algorithm constructs the whole projection and all sample points. Alternatively we can only disable the case in which s turns out to be a satisfying sample. In this case we essentially get a partial CAD in the spirit of Collins and Hong (1991). Note that the proposed techniques are not particularly beneficial for this case, we can however cover this use case with the same implementation.

10.9. Universally Quantified Formulas

We restricted the input to sets of constraints or at least existentially quantified formulas which allowed us to view the CAD as a search procedure for a satisfying sample point. Similarly

a single contradictory sample suffices to prove non-validity of a purely universally quantified formula. In general these two cases can be reduced to each other by negating the formula. Hence we can apply the same modifications to obtain a search procedure for a counterexample for a purely universally quantified formula.

10.10. Equational Constraints

A more recent result of Collins (1998) and McCallum (1999) is that *equational constraints* can be exploited to essentially reduce the projection by one level. They can even be used to obtain further savings as shown in England et al. (2015). This technique can also be applied to our scenario by identifying equational constraints and properly adapting the projection operator.

Note however that the incremental addition and removal of equational constraints may lead to a large number of recomputations, similar to what we describe in Section 10.4. It may be even worse in the case of equational constraints as the insertion of a single constraint removes a potentially large number of projection candidates – that may have been computed already. We are currently exploring efficient schemes to deal with these issues that integrate the use of equational constraints with an incremental CAD.

10.11. Implementation

Even if an abstract algorithm is well understood it can still be a long way to an actual implementation, in particular if one aims for reasonable efficiency. We present a few technical details of our `C++` implementation that may not be immediately obvious.

We use sets of polynomials in several places throughout our algorithms. Unfortunately sets usually are rather inefficient in practice, particularly if we were to actually duplicate the polynomials in those sets. We therefore choose to assign a – per level – unique id to every polynomial which allows us to efficiently store these sets as bitsets, for example taken from *boost::dynamic_bitset*. Note however that we have to take care if we have sets containing polynomials from different levels, for example in Algorithm 14.

There are several known methods to implement a sorted queue, a heap like the standard *C++ std::priority_queue* being the default implementation. Note however that we require some operations that *std::priority_queue* does not support. We want to iterate over all elements or even remove arbitrary elements from the queue during backtracking. While we opted for an extended version of *std::priority_queue*, maintaining a simple list that is regularly sorted also seems like a legitimate solution.

11. Experiments

To effectively estimate the impact of the presented approaches and techniques we present a couple of experiments. All of the following tests were performed – unless stated otherwise – within our own solver SMT-RAT presented in Corzilius et al. (2015). For every input problem the solver binary is executed on an AMD Opteron 6172 processor with a timeout of 60 seconds and at most 4GB of memory. The different variants of the CAD methods are embedded into a SMT solver in SMT-RAT using a SAT solver but no additional preprocessing or theory solving modules. Though additional preprocessing could enhance the overall solver, we felt that it would not help with gauging the relative strengths of the presented approaches.

11.1. Illustrative Example

Before providing benchmarks that actually evaluate the presented techniques we compare our solver SMT-RAT to other well-known tools for CAD computations. We choose the RegularChains library described in Lemaire et al. (2005) and Chen et al. (2009) that is shipped with Maple 2017 by Maplesoft (2017) and QEPCAD B by Brown (2003) as representatives for general purpose computer algebra systems and specialized CAD tools from computer algebra, respectively. The goal of this comparison is to show both the soundness and competitiveness of our solver.

The authors acknowledge that other (and arguably better) tools exist: both Redlog and Mathematica feature CAD implementations and the version of RegularChains we use is outdated. We consider this experiment only a quick check that the overall performance of the underlying methods in SMT-RAT is somewhat on par with other tools. Given that the other tools mostly focus on quantifier elimination and usually do not support the SMT-LIB input format we do not compare against external tools in the more detailed experiments below.

We reconsider the extended example from Figure 5. Computing a full CAD of this example and enumerating all 225 regions takes about 240*ms* using SMT-RAT, 830*ms* using RegularChains in Maple and 70*ms* using QEPCAD B on a desktop computer with an Intel i7-4790K CPU. We consider this to be a confirmation that SMT-RAT does reasonably well performance-wise and the following experiments should provide a meaningful analysis of the general techniques.

11.2. Selection of Benchmarks

Efficiently solving SMT queries is our main motivation for this work, and thus we use the common SMT benchmarks here. The SMT-LIB initiative presented by Barrett et al. (2016) maintains a large and growing repository of sample inputs from a variety of applications. Being solely concerned with nonlinear polynomial arithmetic here, we focus on a logic called `QF_NRA` – quantifier-free nonlinear real arithmetic. At the time of writing in early 2018 `QF_NRA` contained 11354 problem instances from 10 different applications.

It is important to distinguish between a single CAD construction and the solving of a whole SMT problem. Usually solving a single SMT problem yields a whole sequence of calls to the CAD method. The result of each call to the CAD method may be satisfiable or unsatisfiable independent from the overall result. Hence also unsatisfiable problems can benefit from our proposed techniques for CAD computations for satisfiable problems. We thus do not distinguish between satisfiable and unsatisfiable SMT problems here. If the particular solver gives a (correct) result after at most 60 seconds and uses at most 4GB of memory we consider the example *solved* and otherwise *unsolved*.

We refrain from an analysis of the individual benchmark groups and acknowledge that they might very well contain some bias towards specific problem structures. For example a significant number of benchmarks are unsatisfiable considering only the Boolean abstraction, thus suggesting that doing any theory computation is just overhead. On the other hand a theory solver like CAD might be of great assistance if it can determine unsatisfiability for very small sets of constraints and thereby reduce the Boolean search space on these examples. Ultimately this only highlights another class of problems for which more specialized heuristics could prove to be valuable.

Finally we want to note that a performance increase by a certain percentage usually does not yield the same percentage of newly solved problem instances. When ordering the problems by their *difficulty* – whatever it is that makes a particular problem difficult for a particular solver

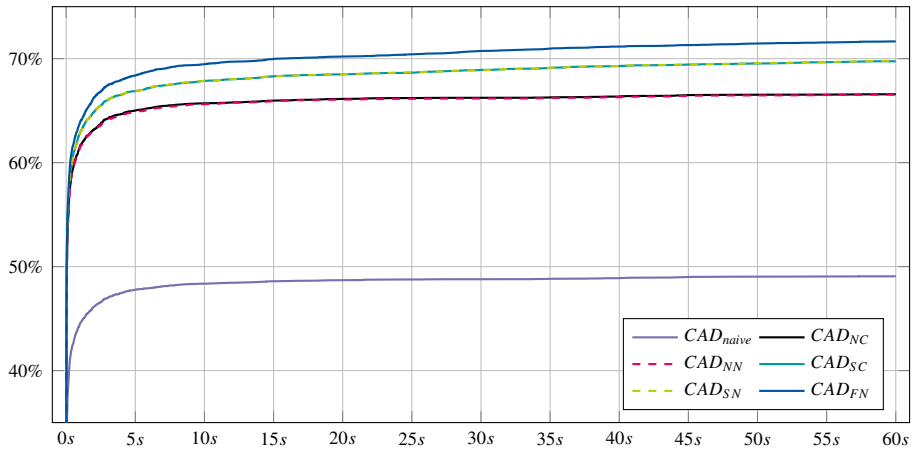


Figure 8: Percentage of instances solved within a given time.

– the runtime usually grows exponentially. At the same time every change to any heuristic solves a few instances but fails to solve some that were solved previously. A very small increase or decrease in the number of solved instances is probably not meaningful but the result of a statistical fluctuation. This also implies that any metric involving the absolute runtimes should be considered with caution, for example solving more problems overall almost inevitably leads to a higher average runtime.

11.3. Different Versions of Incrementality

We presented different ways to compute a CAD incrementally and how to perform backtracking. As a baseline for comparison we use a solver CAD_{naive} with a naive CAD implementation that computes a full CAD from scratch in every theory call. In comparison CAD_{NC} and CAD_{NN} retain the computed CAD from the last theory call and merely update it to a full CAD every time. They use *chronological* and *non-chronological* backtracking, respectively. CAD_{SC} and CAD_{SN} use the approach we called *simple incrementality*, again with *chronological* and *non-chronological* backtracking. Finally we also implemented CAD_{FN} that uses *full incrementality* with *non-chronological* backtracking.

Table 1a and Figure 8 provide an overview of the results for these solvers. We can see that CAD_{naive} already solves a fair amount of examples. With CAD_{NC} or CAD_{NN} the number of solvable instances grows significantly, thanks to retaining the state during multiple calls to the CAD method.

Using CAD_{SC} and CAD_{SN} results in another notable performance increase as we can avoid considering constraints in many cases. CAD_{FN} proves to be better once again as it apparently avoids many projection steps in practice. We consider this result a confirmation that the effort to implement the proposed techniques is worthwhile.

Interestingly the impact of using either chronological or non-chronological backtracking is negligible. We conclude that our SAT solver rarely behaves in a way that leads to non-chronological backtracking in the theory solver or using Algorithm 13 is rather cheap in these cases. One might have expected this as most problem instances from `QF_NRA` have almost no Boolean structure which would be needed to allow for non-chronological backtracking to occur.

Solver	solved		runtime
CAD_{naive}	5571	49.1 %	0.69
CAD_{NN}	7555	66.5 %	0.64
CAD_{NC}	7559	66.6 %	0.60
CAD_{SC}	7919	69.7 %	1.08
CAD_{SN}	7924	69.8 %	1.11
CAD_{FN}	8158	71.9 %	1.22

(a) Results for incrementality and backtracking

Solver	solved		runtime
CAD_{SR}	8144	71.7 %	1.20
$CAD_{S\infty}$	8146	71.7 %	1.21
CAD_{SC}	8147	71.8 %	1.21
CAD_{S0}	8154	71.8 %	1.20
CAD_{SI}	8155	71.8 %	1.19
CAD_{SL}	8158	71.9 %	1.22

(b) Different sampling heuristics

Solver	solved		runtime
$CAD_{\texttt{LTSA}}$	8118	71.5 %	1.21
$CAD_{\texttt{LS}}$	8121	71.5 %	1.22
$CAD_{\texttt{LT}}$	8138	71.7 %	1.20
$CAD_{\texttt{LLTS}}$	8143	71.7 %	1.22
$CAD_{\texttt{LLT}}$	8144	71.7 %	1.20

(c) Different lifting orders

Solver	solved		runtime
$CAD_{\texttt{PSC}}$	8074	71.1 %	1.13
$CAD_{\texttt{PC}}$	8075	71.1 %	1.13
$CAD_{\texttt{PPC}}$	8076	71.1 %	1.12
$CAD_{\texttt{PLC}}$	8135	71.6 %	1.28
$CAD_{\texttt{P1C}}$	8135	71.6 %	1.18

(d) Different projection orders

Table 1: Experimental results on all 11354 benchmarks from `QF_NRA`.
Number and percentage of solved instances with average runtime in seconds on solved instances.

11.4. Different Heuristics

The proposed techniques contain multiple places where custom heuristics can be employed: which sample to choose in Algorithm 3, the order in which to consider sample points for lifting or which polynomial to use for lifting a specific sample point in Algorithm 5 and the projection order in Algorithm 9 and Algorithm 11. We now present several heuristics for each of these and evaluate how they affect the overall performance based on CAD_{FN}. Note that for every comparison we only vary one and select fixed heuristics for all the others and hence the results should not be compared across multiple tables.

We first check different ways to generate sample points from an interval between two roots. Our assumption is that it makes sense to select values that are easy to calculate with, ideally integers. The heuristics we tried are using the midpoint (CAD_{SC}) or an integer value close to the midpoint (CAD_{SI}), the smallest or largest integer in the interval (CAD_{SL} and CAD_{SR}) and an integer close to zero or far away from zero (CAD_{S0} and $CAD_{S\infty}$). The results shown in Table 1b demonstrate that these heuristics can make a difference in some rare cases, but are generally not significant.

Another heuristic is the lifting order of the samples in $L.Q$. Essentially we try to prefer certain samples that quickly lead to a satisfying sample point. We identified the following criteria to distinguish sample points: *absolute value*, *level*, *size* and *type* that we denote by A, L, S and T, respectively. We use some approximation of the size of the internal representation for the *size* and consider the three different *types* integral, rational and algebraic. These criteria were combined in a lexicographical manner with the results shown in Table 1c. Once again we see that the differences are rather small on average. Note that all orders that use the level as a criterion are strongly coupled to the variable ordering that is used and the results may thus change drastically if the variable variable order changes.

The *projection order* governs in which order polynomials are considered during the projection in Algorithm 9 and Algorithm 11. Again we identified several criteria, in this case the *level* of the projection candidate, whether it is a *single* or *paired* projection step and a heuristic measure for the *complexity* of the involved polynomials. We denote these by `l` and `L` for the level in *decreasing* and *increasing* order, `S` and `P` for single and paired and `C` for complexity, respectively. Table 1d shows some experimental results for different projection orders. Similar to the lifting order, all orders that use the level as a criterion are dependent on the variable order. It turns out to be beneficial to consider the level of projection candidates and surprisingly both increasing and decreasing order are an improvement compared to not considering the level at all.

12. Conclusion and Future Work

Based on a restriction of the cylindrical algebraic decomposition to fully existentially quantified formulas we regard CAD as a search procedure for a satisfying sample point. The traditional CAD is essentially split into many small steps such that we can pause and continue the work after every step. We motivate the solving of a sequence of such problems in an incremental fashion and argue why a proper backtracking mechanism is crucial in practice.

Starting from the *non-incremental* CAD two incremental variants are presented that we call *simple incremental* and *full incremental*. This is extended by *backtracking* to allow for the removal of constraints, either *chronologically* or *non-chronologically*.

We then present data structures and algorithms that implement those approaches and are combined to a complete CAD implementation. Afterwards we briefly discuss a list of extensions that may be of interest to the reader and conclude with an experimental evaluation using our solver SMT-RAT checking a number of different heuristics.

We argue that some degree of incrementality is highly beneficial in a scenario that only requires a satisfying sample point instead of a full CAD. We show that the different heuristics need more investigation, though some of them do not seem to have a great effect. Nevertheless any of the heuristics might have a significant impact on a particular class of examples in practice and their interactions are not well understood yet.

Our future plans include the creation of a framework that unifies the backtracking mechanism with the optimizations mentioned in Section 10.4 and Section 10.10 in a more efficient way. Furthermore we suspect that another variable ordering could have a great impact as the existing variable orderings are designed with respect to a single formula while we consider a sequence of different, though similar, formulas. For our scenario we may adapt the existing variable ordering or even change the order dynamically.

References

Barrett, C., Fontaine, P., Tinelli, C., 2016. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`.

Barrett, C. W., Sebastiani, R., Seshia, S. A., Tinelli, C., et al., 2009. Satisfiability modulo theories. Handbook of satisfiability, 825–885.

Bradford, R., Davenport, J. H., England, M., McCallum, S., Wilson, D., 2016. Truth table invariant cylindrical algebraic decomposition. Journal of Symbolic Computation 76, 1 – 35.
URL `http://www.sciencedirect.com/science/article/pii/S0747717115001005`

Brown, C. W., 2001. Improved projection for cylindrical algebraic decomposition. Journal of Symbolic Computation 32 (5), 447 – 465.

Brown, C. W., 2003. QEPCAD B: a program for computing with semi-algebraic sets using CADs. ACM SIGSAM Bulletin 37 (4), 97–108.

Caviness, B. F., Johnson, J. R., 1998. Quantifier elimination and cylindrical algebraic decomposition. Springer Science & Business Media.

Chen, C., Moreno Maza, M., Xia, B., Yang, L., 2009. Computing cylindrical algebraic decomposition via triangular decomposition. In: Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation. ISSAC '09. ACM, New York, NY, USA, pp. 95–102.
URL http://doi.acm.org/10.1145/1576702.1576718

Collins, G. E., 1975. Quantifier elimination for real closed fields by cylindrical algebraic decompostion. In: Automata Theory and Formal Languages 2nd GI Conference Kaiserslautern. Springer, pp. 134–183, reprint in Caviness and Johnson (1998).

Collins, G. E., 1998. Quantifier elimination by cylindrical algebraic decomposition – twenty years of progress. In: Quantifier elimination and cylindrical algebraic decomposition. Springer, pp. 8–23.

Collins, G. E., Hong, H., 1991. Partial cylindrical algebraic decomposition for quantifier elimination. Journal of Symbolic Computation 12 (3), 299–328.

Collins, G. E., Loos, R., 1983. Real Zeros of Polynomials. Springer Vienna, Vienna, pp. 83–94.

Corzilius, F., Kremer, G., Junges, S., Schupp, S., Ábrahám, E., 2015. SMT-RAT: an open source C++ toolbox for strategic and parallel SMT solving. In: International Conference on Theory and Applications of Satisfiability Testing. Springer, pp. 360–368.

Coste, M., Roy, M., 1988. Thom's lemma, the coding of real algebraic numbers and the computation of the topology of semi-algebraic sets. Journal of Symbolic Computation 5 (1), 121 – 129.

Dolzmann, A., Seidl, A., Sturm, T., 2004. Efficient projection orders for CAD. In: International Symposium on Symbolic and algebraic computation. ACM, pp. 111–118.

England, M., Bradford, R., Davenport, J. H., 2015. Improving the use of equational constraints in cylindrical algebraic decomposition. In: Proceedings of the 2015 ACM on International Symposium on Symbolic and Algebraic Computation. ISSAC '15. ACM, New York, NY, USA, pp. 165–172.
URL http://doi.acm.org/10.1145/2755996.2756678

England, M., Bradford, R., Davenport, J. H., Wilson, D., 2014. Choosing a variable ordering for truth-table invariant cylindrical algebraic decomposition by incremental triangular decomposition. In: Hong, H., Yap, C. (Eds.), International Congress on Mathematical Software. Springer, pp. 450–457.

Hentze, W., 2017. Computing minimal infeasible subsets for the cylindrical algebraic decomposition. Bachelor's thesis, RWTH Aachen University, available at https://ths.rwth-aachen.de/theses/.

Huang, Z., England, M., Wilson, D., Davenport, J. H., Paulson, L. C., Bridge, J., 2014. Applying machine learning to the problem of choosing a heuristic to select the variable ordering for cylindrical algebraic decomposition. In: Intelligent Computer Mathematics. Springer, pp. 92–107.

Jaroschek, M., Dobal, P. F., Fontaine, P., 2015. Adapting real quantifier elimination methods for conflict set computation. In: International Symposium on Frontiers of Combining Systems. Springer, pp. 151–166.

Jovanović, D., De Moura, L., 2012. Solving non-linear arithmetic. In: International Joint Conference on Automated Reasoning. Springer, pp. 339–354.

Kremer, G., Corzilius, F., Ábrahám, E., 2016. A generalised branch-and-bound approach and its application in SAT modulo nonlinear integer arithmetic. In: Gerdt, V. P., Koepf, W., Seiler, W. M., Vorozhtsov, E. V. (Eds.), Computer Algebra in Scientific Computing. Springer, pp. 315–335.

Lazard, D., 1994. An improved projection for cylindrical algebraic decomposition. In: Algebraic Geometry and its Applications. Springer, pp. 467–476.

Lemaire, F., Moreno Maza, M., Xie, Y., 2005. The regularchains library in maple. ACM SIGSAM Bulletin 39 (3), 96–97.

Loup, U., Scheibler, K., Corzilius, F., Ábrahám, E., Becker, B., 2013. A symbiosis of interval constraint propagation and cylindrical algebraic decomposition. In: International Conference on Automated Deduction. Springer, pp. 193–207.

Maplesoft, 2017. Maple 2017. Waterloo Maple Inc., Waterloo, Ontario.

McCallum, S., 1998. An improved projection operation for cylindrical algebraic decomposition. In: Quantifier Elimination and Cylindrical Algebraic Decomposition. Springer, pp. 242–268.

McCallum, S., 1999. On projection in cad-based quantifier elimination with equational constraint. In: Proceedings of the 1999 international symposium on Symbolic and algebraic computation. ACM, pp. 145–149.

Neuhäuser, T., 2018. Quantifier elimination by cylindrical algebraic decomposition. Bachelor's thesis, RWTH Aachen University.

Viehmann, T., Kremer, G., Abraham, E., 2017. Comparing different projection operators in the cylindrical algebraic decomposition for SMT solving. In: International Workshop on Satisfiability Checking and Symbolic Computation. CEUR-WS 1974.